UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Máster Universitario en Ingeniería de Telecomunicaciones

TRABAJO DE FIN DE MÁSTER

Contribuciones a la simulación de sistemas de vídeo-seguridad con múltiples cámaras

Vinicio David Pazmiño Moya
Tutor: Juan Carlos San Miguel Avedillo

Septiembre, 2020

Contribuciones a la simulación de sistemas de vídeo-seguridad con múltiples cámaras

Vinicio David Pazmiño Moya
Tutor: Juan Carlos San Miguel Avedillo

Video Processing and Understanding Lab
Departamento de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Septiembre, 2020

Trabajo parcialmente financiado por el Ministerio de Economía y Competitividad de Gobierno de España bajo el proyecto MobiNetVideo TEC2017-88169-R del VPULab

Resumen

La implementación de un sistema orientado al estudio de la visión artificial es interesante debido a la relevancia que éste puede tener sobre el desarrollo de simulaciones, lo cual permite crear escenas y eventos que contribuyan con la mejora continua de código y algoritmos dentro de la visión artificial. Partiendo de la API (*Application Programming Interface*) del simulador llamado: *Multi-camera System Simulator* (MSS) desarrollado en el TFG de 2017 por Mario González dentro del laboratorio de investigación VPULab de la Universidad Autónoma de Madrid, el presente Trabajo de Fin de Máster (TFM) incorpora mejoras en el simulador para potenciar su funcionalidad. El proyecto proporciona actualizaciones en las librerías, nuevas funciones y una API de cliente desarrollada en Python (v3.6.9). Este trabajo hace uso y manejo de múltiples cámaras virtuales implementadas en una primera parte por el diseñador del escenario y por otro lado mediante una API de usuario que incorpora un amplio rango de funcionalidades para el manejo del sistema con múltiples cámaras. La herramienta permite la configuración de las cámaras y su uso tanto en remoto como en *localhost* con una estructura cliente-servidor. Además, en el documento se detalla el diseño, la implementación, la integración y los cambios necesarios para generar ilimitados entornos de prueba. Por último, se evalúa el rendimiento del sistema desarrollado y se discuten los resultados, estableciendo configuraciones recomendadas y descubriendo los límites técnicos del simulador, así también se realiza una evaluación del comportamiento de los recursos computacionales al ejecutar el simulador.

Palabras clave

Multi-cámara, Unity 3D, simulación, escenarios, Visión Artificial, Mapa Semántico, Interfaz de Programación de Aplicaciones (API)

Abstract

The implementation of a system oriented to the study of artificial vision is interesting due to the relevance it can have on the development of simulations, which allows the creation of scenes and events that contribute to the continuous improvement of code and algorithms within artificial vision. Based on the API (Application Programming Interface) of the simulator called: Multi-camera System Simulator (MSS) developed in the TFG of 2017 by Mario González within the VPULab research laboratory of the Universidad Autónoma de Madrid, the present End of Master's Work (TFM) incorporates improvements in the simulator to enhance its functionality. The project provides library updates, new functions and a client API developed in Python (v3.6.9). This work makes use and handling of multiple virtual cameras implemented in the first part by the scenario designer and on the second part through a user API that incorporates a wide range of functionalities for handling the system with multiple cameras. The tool allows the configuration of the cameras and their use both remotely and locally with a client-server structure. In addition, the document details the design, implementation, integration and changes required to generate unlimited test environments. Finally, the performance of the developed system is evaluated and the results are discussed, establishing recommended configurations and discovering the technical limits of the simulator, as well as an evaluation of the behaviour of the computer resources when the simulator is executed.

Keywords

Multi-camera, Unity 3D, simulation, scenarios, Artificial vision, Semantic map, Application Programming Interface (API)

Agradecimientos

Quiero agradecer primero a Dios por haberme permitido cumplir con un nuevo objetivo en mi vida y culminar exitosamente otra etapa totalmente diferente de mi vida, gracias a lo cual he aprendido mucho como profesional y ser humano. También lo dedico a todo el amor que me han dado mis padres, Marco y Mery, por su apoyo incondicional en todo momento, gracias a ellos estoy aquí logrando un nuevo y mejor futuro, y es posible gracias a su amor y guía. Quiero agradecer también muy especialmente a mi familia, tía y primos; Ivan, Margodt, Flerida, Cinthya y Jonathan de quines día a día he podido encontrar un refugio y un apoyo incondicional en estos dos años fuera de casa.

También quiero agradecer a la Universidad Autónoma de Madrid y a la Escuela Politécnica Superior por brindarme la posibilidad de estudiar aquí, a su vez quiero agradecer a mi profesor y tutor del TFM Juan Carlos San Miguel Avedillo por su ayuda y guía en todo este proceso, por la confianza depositada para desarrollar y llevar a cabo lo mejor de mi en este proyecto por el cual pude pertenecer al grupo de investigación VPU donde aprendí y desarrolle nuevos conocimientos en mi preparación profesional.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Motivación

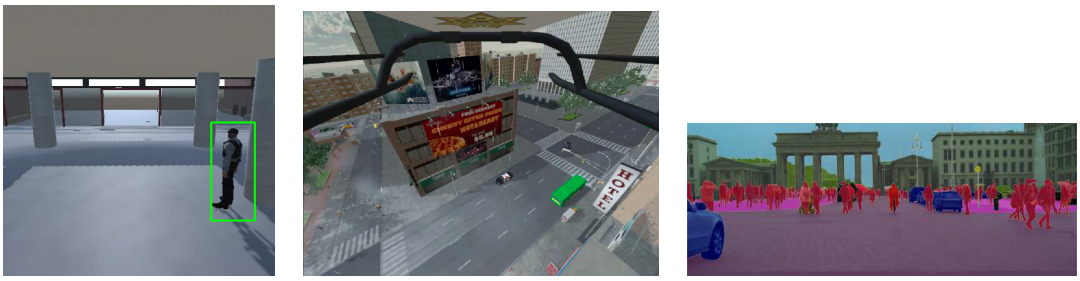
La última década ha traído consigo una evolución importante en cuanto al desarrollo de las telecomunicaciones. El crecimiento exponencial en la investigación tecnológica ha permitido dar un gran salto para resolver problemas complejos en distintas áreas de conocimiento. Una de las soluciones es la visión artificial, que con el desarrollo de herramientas potentes en hardware permiten implementar algoritmos y métodos de procesamiento mucho más eficientes. La visión artificial es un conjunto de técnicas y metodologías que permiten adquirir, procesar, analizar y comprender mediante imágenes el mundo real. En teoría, se quiere conseguir que la máquina pueda replicar las decisiones que la visión humana pueda tomar al observar cualquier conjunto de datos.

Las herramientas y algoritmos de la visión artificial ha impulsado a la creación de múltiples aplicaciones y servicios que las empresas pueden ofrecer sobre dispositivos móviles, cámaras de vídeo, drones entre otras. Esto conlleva al mejoramiento de resultados, eficiencia, velocidad, estabilidad, experiencias inmersivas 3D y calidad de imagen, lo que supone un valor añadido en los servicios. [1]. Los simuladores actualmente brindan calidad para replicar múltiples escenarios y una variedad de experimentos con eventos y situaciones específicos. Estos simuladores son extremadamente útil para poder evaluar cualquier algoritmo en desarrollo y/o entrenar redes neuronales complejas. Las pruebas exhaustivas deben ser una parte integral y funcional en cualquier proceso de desarrollo para reducir posibles fallas en el mundo real, por lo que los simuladores se convierten en una herramienta de vital importancia en el área de la visión artificial [2], [3].

Una herramienta de simulación trae consigo muchas ventajas para entrenar adecuadamente cualquier algoritmo ya que es posible recrear escenarios controlados y generar resultados a medida. En la visión artificial se necesita analizar una gran cantidad de datos y a través de un simulador se puede controlar y gestionar distintos parámetros (cantidad de eventos, generación aleatoria de personas, peatones o transeúntes, densidad de vehículos en la escena, creación de cámaras de vídeo tanto en posición como en cantidad) [4]. Esto permite generar tantas escenas como requiera el usuario para mejorar un algoritmo en particular [5], [6].

Actualmente los simuladores son herramientas muy potentes por lo que en el mercado existe una multitud de programas que permiten hacer simulaciones. En el artículo *Software Laboratory*

for Camera Networks Research [2] se recrea un escenario 3D con gran afluencia de personas donde una red de cámaras a gran escala es capaz de hacer identificación y seguimiento de peatones. Este trabajo es un indicativo de la importancia de poder manipular distintos parámetros de una escena para obtener buenos resultados al aplicar algoritmos de visión artificial [7].



(a) Desarrollo de simulador (b) Variedad de Escenarios (c) Segmentación Semantica

Figura 1.1: Aplicaciones de *Computer Vision* [7], [4], [8] .

La fiabilidad en cuanto a calidad de imagen y vídeo es un indicativo de que un simulador es competitivo en el mercado. Cuando se quiere generar secuencias de escenarios sintéticos [9] es muy importante imitar con gran precisión y detalle la condiciones del mundo real. Un simulador con buenas prestaciones permite un renderizado óptimo de una red de cámaras para obtener datos (*frames*) con los cuales se pueda trabajar en mejora de algoritmos existentes en la visión artificial.

La videovigilancia pertenece a un campo muy extendido de la visión artificial mediante el cual se procesa y analiza señales extraídas en una transmisión de vídeo, dentro de este proceso es importante el reconocimiento de patrones y la variedad de cámaras colocadas en puntos concretos de un entorno, por lo que es importante conocer el desarrollo reciente de tecnologías orientadas a este ámbito multidisciplinar, conocimientos relevantes como calibración de cámaras, topologías, seguimiento de objetos y el uso de tipos de cámaras activas o estáticas permiten generar escenarios eficientes y actualizados para mejorar la precisión en la búsqueda de soluciones [10], [11].

Otro de los grandes objetivos en la visión artificial es la conducción autónoma [12], área de investigación que estudia las redes neuronales y aprendizaje profundo (*deep learning*). En esta rama el enfoque principal es desarrollar escenas de tráfico proporcionando datos cercanos a la realidad. Así la red pueda ser entrenada y posteriormente aplicar las secuencias de vídeo para comprobar la fiabilidad del algoritmo implementado [13]. Sin embargo la importancia de estos simuladores no solo recaen en generar datos sintéticos sino en la posibilidad de construir escenas virtuales donde es posible navegar en primera persona. Debido a lo cual se implementa *scripts* que se acoplen a las necesidades de los usuarios con cualquier tipo de experiencia en la visión artificial o redes profundas [14].

La conducción autónoma hace unos años se pensaba improbable debido a la complejidad de todas las variables que implica manejar un vehículo. A pesar de la problemática el avance tecnológico muestra que es posible tener autos parcialmente autónomos. Detrás de todo esto, una herramienta fundamental para su funcionamiento es la segmentación semántica, elemento clave

para entrenar redes neuronales. Este tipo de estudios ya se lleva a cabo y es necesario obtener una gran cantidad de datos para aplicar una clasificación y etiquetado semántico de escenas a nivel de píxel [15], [8]. Aquí es donde un simulador que entregue dichas características es una herramienta fundamental para facilitar el entrenamiento de estos algoritmos en la industria automotriz.

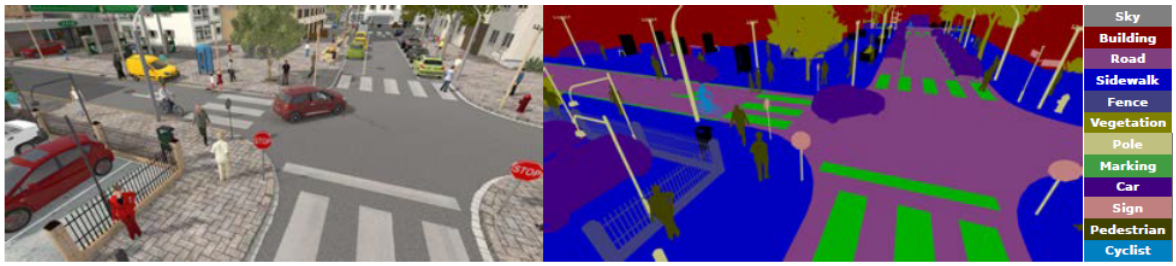


Figura 1.2: Clasificación Semántica y etiquetado de clases [15]

1.1.1 Clasificación semántica

Dentro de la importancia de la visión artificial es la tarea de la clasificación y reconocimiento de objetos dentro de una imagen. A diferencia de estos sistema de detección la segmentación semántica va un paso mas allá en identificar los objetos de una escena. El objetivo principal de la segmentación semántica es otorgar una etiqueta o categoría a cada píxel de una imagen como se puede observar en la Figura 1.3.

Figura 1.3: Segmentación semántica: Identificar, clasificar y etiquetar de un color cada objeto en la imagen [16]

En la visión artificial uno de los problemas que se más se estudia es la segmentación de imágenes en píxeles, la tarea de implementar un mapa semántico es la de clasificar cada píxel de una imagen por lo cual el uso de redes neuronales convoluciones profundas es una de las herramientas actuales más robustas para tratar dicha área. Para la segmentación semántica es importante tener en cuenta que al clasificar píxeles, estos deben pertenecer a una determinada clase, es decir que dentro de este campo se define un conjunto predefinido de varias clases en donde van a pertenecer cada píxel de la imagen en cuestión.

La importancia de implementar dentro del simulador la opción de añadir un mapa semántico a la escena es obtener un etiquetado claro y rápido en los vídeos generados por las cámaras del simulador. Es decir que al clasificar todos los objetos de la escena se puede obtener una gran cantidad de datos con los que se puede entrenar una red neuronal para probar los algoritmos dedicados a la segmentación semántica.

1.1.2 Segmentación por clases

La segmentación semántica es el campo de la visión artificial para cumplir la tarea de identificar los limites de un objeto a nivel de píxel. Gracias a esto el clasificar las diferentes partes de una imagen en clases es el principal objetivo para obtener una información interpretable. En la segmentación semántica cada clase debe tener un significado en el mundo real, por ejemplo tomar todos los píxeles de una imagen que pertenecen a un peatón y marcarlos con un color perteneciente a la clase "persona".

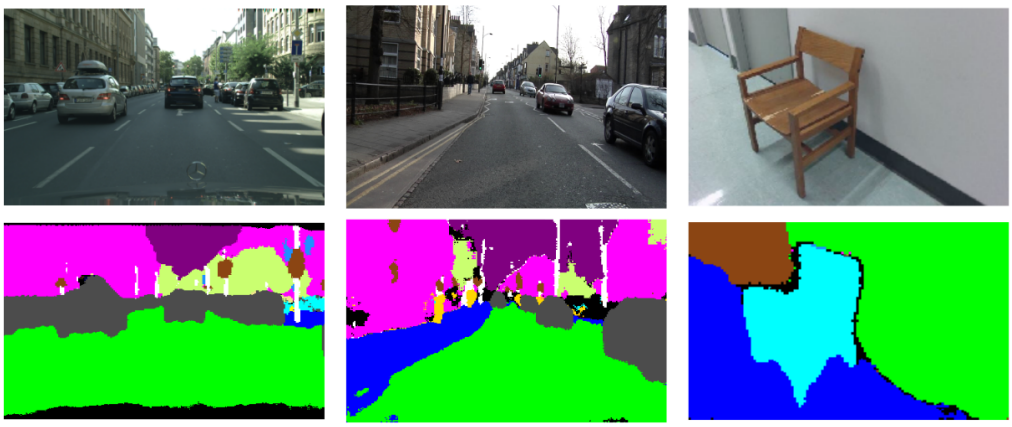


Figura 1.4: Predicción con ENet en diferentes *datasets* (Cityscapes | CamVid | SUN) [17].

La segmentación semántica permite lograr una comprensión con mucho más detalle de imágenes que simplemente clasificar imágenes o detectar objetos en una foto. Es en esta área de la visión artificial que se derivan una gran cantidad de aplicaciones como lo es la conducción autónoma y en lo que se ha puesto parte de este proyecto en la monitorización y vigilancia de cámaras de tránsito o redes de multi-cámras dentro de un escenario de ciudad.

El Trabajo Fin de Máster se desarrolla bajo el contexto de la asignatura de "Sistemas de Vídeo-Seguridad", área donde se realizan simulaciones en el despliegue de un sistema de múltiples cámaras para la vídeo-seguridad de distintos entornos con el simulador MSS [7]. De estos se destaca la Escuela Politécnica Superior (EPS) y una ciudad completa desarrollada en Unity. El simulador MSS [7] contiene distintos escenarios fácilmente personalizables donde se puede posicionar cámaras variando tanto sus parámetros (fps, resolución, mapa semántico, cámara estática, cámara PTZ, etc.) como su ubicación y posición dentro de la escena. MSS proporciona un flujo de *frame* para ser transmitidos y visualizados en pantalla mediante una comunicación cliente-servidor.

1.2 Objetivos

El objetivo de este trabajo Fin de Máster es diseñar e implementar funcionalidades y mejoras al simulador multi-cámara de vídeo-seguridad basado en Unity mediante el desarrollo de escenarios que incorporan nuevas características a las cámaras en el servidor, y renovando la API del cliente de manera que se obtengan mayores prestaciones en la investigación de algoritmos de la visión artificial.
Para lo cual en concreto se definen los siguientes sub-objetivos:

- Estudiar la estructura del simulador previamente desarrollado mediante simples test de ejecución para estructurar el tipo de funciones que serán incorporadas en la nueva versión del simulador.
- Comprender el funcionamiento de Unity como servidor para proveer el sistema multi-cámara mediante el cual se adecue nuevos *scripts* para que se envíe, desde el cliente, los nuevos parámetros a las cámaras puestas en escena.
- Desarrollar una nueva versión de API en el cliente con Python 3.X para que se apliquen algoritmos recientes de visión artificial mayormente desarrollados en éste lenguaje de programación para el procesado y análisis de vídeo.
- Implementar nuevas características en el simulador tales como: cámaras de escenario, cámaras móviles, estáticas y/o PTZ, transmisión de *frames* bajo demanda, capa semántica de renderización, mediante una actualización de librerías del simulador en Unity para que el usuario las despliegue según la necesidad y utilidad para el tratamiento de imágenes.
- Desarrollar varias opciones de escenarios incorporando objetos pre-fabricados (*prefabs*) y tipos de entornos (simulación de tráfico, capa semántica) para la preparación de experimentos y la validación de rendimiento del simulador en términos de recursos computacionales necesarios para su ejecución.
- Diseñar un protocolo para la realización de experimentos mediante análisis de imágenes sobre datos del simulador para la obtención resultados sobre el trabajo.

1.3 Organización de la memoria

En el presente documento se puede encontrar la siguiente estructura:

- Capítulo 1. Donde se puede observar la motivación y los objetivos para el desarrollo del Trabajo Fin de Máster.
- Capítulo 2. El cual se puede encontrar una descripción general de alternativas de escenarios y el trabajo previo del simulador MSS con la arquitectura base de funcionamiento.
- Capítulo 3. El cual se presenta la implementación de mejoras en la API servidor, incorporando métodos para el acceso remoto de cámaras en escena y añade la funcionalidad de visualizar el *ground-truth* de la segmentación semántica en imágenes.
- Capítulo 4. Describe el desarrollo, funcionamiento y ejecución de la API desarrollada en Python y los ejemplos que se pueden simular.

- Capítulo 5. Donde se presenta un protocolo para desarrollar la estructura de experimentos y se analiza los resultados de evaluación de distintas pruebas.
- Capítulo 6. Donde se describe y presenta una aplicación se segmentación semántica con algoritmos desarrollados en la visión artificial.
- Capítulo 7. Apartado donde se presentan los logros alcanzados en el trabajo y posibles líneas de investigación relacionados con el simulador.
- Bibliografía.

Adicionalmente se estructura con distintos anexos complementarios con un mayor detalle y explicaciones.

- Apéndice A. Este anexo describe la configuración y preparación del entornar cliente para ejecutar la API de Python.
- Apéndice B. Donde se detalla algunas consideraciones de código importantes cuando se usa distintas versiones de Python (2.x vs 3.x) dentro de la API cliente.
- Apéndice C. Este apartado incorpora el código principal de Matlab para procesar los datos y generar las gráficas para evaluar los experimentos del simulador.
- Apéndice D. Este anexo se explica a detalle el código seleccionado para la aplicación de segmentación semántica sobre fotogramas de evaluación.

Capítulo 2

Estado del arte

Este capítulo resume las diferentes alternativas que existen en motores gráficos 3D en el desarrollo de escenas reales y compara algunas de las mas importantes, seguidamente se estudia el funcionamiento básico del simulador MSS [7] y su estructura general.

2.1 Motores gráficos

Existen diversos motores gráficos para crear simuladores virtuales con características diferenciadas, tal que los desarrolladores pueden personalizar y recrear cualquier escena sin limitaciones.

2.1.1 Unity

La industria de los videojuegos es la principal área en el desarrollo de escenarios 3D donde es posible crear mundos virtuales para poder experimentar y sintetizar una gran variedad de eventos para la recreación de simulaciones complejas, esto mediante el avance tecnológico permite aplicar algoritmos de física clásica, choque de cuerpos e Inteligencia artificial (IA) con una mayor precisión en la toma de decisiones computacionales guiadas por la visión artificial.

Figura 2.1: Paquete de recursos para escenario del simulador[1]

El simulador *Multi-camera System Simulator* desarrollado en 2017 [7], es una herramienta de software donde es posible analizar vídeo y eventos creados artificialmente para plasmar un caso de estudio basado en simulación [11], [12] con la utilización de OpenCV y Visual Studio para entrar en el mundo de visión artificial. A través de este sistema se pretende solventar las limitaciones técnicas existentes en el mundo real, la plataforma de Unity brinda un motor de desarrollo gráfico 3D [18] con amplia gama de posibilidades para crear distintos entornos, permitiendo de tal modo generar escenarios de prueba flexibles para el análisis de algoritmos basados en IA (inteligencia Artificial). En la actualidad existe una gran comunidad de desarrolladores software que año tras año actualizan todo tipo de librerías dentro de las cuales Unity no es la excepción y es muy fácil encontrar todo tipo de aplicativos, se puede tomar como referencia el proyecto fin de carrera [19] que presenta un manual muy estructurado para el desarrollo de videojuegos donde se aprende a crear y estructurar un proyecto en Unity, al igual que manipular la interfaz gráfica intuitiva y programación de *scripts* para conocer el mundo de posibilidades que ofrece el motor de desarrollo 3D, herramienta fundamental en la implementación de nuevas propuestas de proyectos.

2.1.2 Alternativas a Unity

Algunos de las alternativas de motores gráficos distintas a Unity se en listan a continuación:

1. **Panda 3D.-**[2] Motor gráfico de código abierto y gratuito que es posible desplegar en plataformas como Windows, Mac OS y Linux.

2. **Ogre 3D.-**[3] Motor gráfico que proporciona un SDK en el lenguaje de C++ y puerto para C#, permite el desarrollo de videojuegos lo que representa una herramienta importante para crear escenas.

3. **Quest 3D.-**[4] Este motor gráfico 3D es una herramienta centrada en la visualización arquitectónica por lo que no contiene muchas características para el desarrollo de juegos como otras existentes en el mercado. Una de sus principales desventajas es que tiene limitadas las funciones de renderización y no soporta *streaming*.

4. **ShiVa 3D.-**[5] Este motor gráfico es muy comercial ya que soporte muchas mas plataformas que sus competidores, además tiene soporte para un reproductor web. Su edición posee un conjunto de características amplio y soporta *streaming*.

5. **Torque 3D.-**[6] Es un motor gráfico compatible con los principales sistemas operativos e incluye un paquete de física de cuerpos para simular escenarios mas realistas, además posee funciones avanzadas de renderización e iluminación.

6. **Source Engine.-**[7] Desarrollado por la empresa Valve Corporation en 2004 con el conocido juego Counter-Strike. Esta herramienta con el tiempo ha evolucionado y escalando para tener mayores prestaciones en el desarrollo de entornos virtuales. En su haber posee

[1] https://goo.gl/trenoj
[2] https://www.panda3d.org/
[3] https://www.ogre3d.org/
[4] https://web.archive.org/web/20080313120524/http://quest3d.com/
[5] https://www.saphirprod.com/
[6] http://www.garagegames.com/products/torque-3d
[7] https://developer.valvesoftware.com/wiki/Source_Engine_Features

desarrollo avanzado de luces y sombras lo que brinda a la escena una imagen mucho más realista.

7. **UnrealCV.-**[8] Es un proyecto de simulación que se enfoca en la visión artificial, con el objetivo de construir ambientes virtuales basado en código abierto. Hace uso del motor gráfico *Unreal Engine* 4 (UE4). La mayor de sus desventajas es el no poder transmitir imágenes en tiempo real ya que solo se almacena la información en memoria externa [4].

En la Figura 2.2 se puede observar distintas imágenes tomadas de ejemplos de juegos y simulaciones desarrolladas mediante los motores gráficos descritos en este apartado.

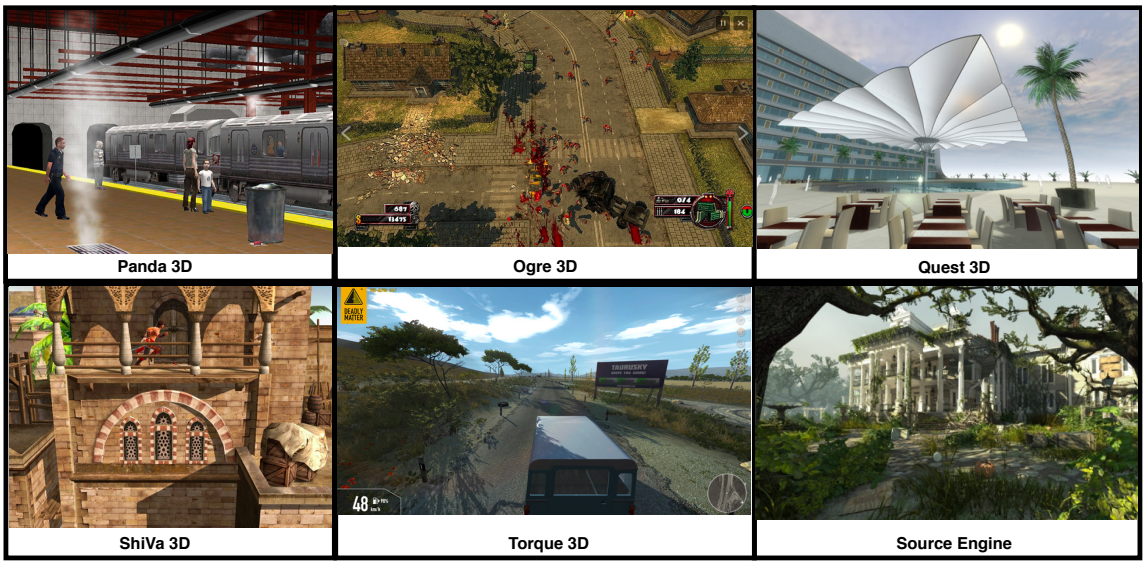


Figura 2.2: Entornos desarrollados con distintos motores gráficos 3D [4].

2.1.3 Comparativa

En la Tabla 2.1 se resume las principales diferencias entre distintos motores gráficos para el desarrollo del simulador, donde se observa que existen gran variedad de herramientas en el mercado, sin embargo el decantarse por la opción de Unity brinda una a amplia la documentación y soporte para los desarrolladores.

Una versión inicial del simulador fue desarrollada con un escenario básico donde se prueba principalmente el funcionamiento de la red mult-cámara, en éste se modela el edificio principal de la Escuela politécnica Superior (EPS) "Alan Turing" [7]. Posteriormente con la gran cantidad de recurso disponibles en el *Aseet Store* de Unity se ha trasladado el simulador sobre *Modern City Pack*, el cual contiene una variedad de recursos con latas prestaciones (ver Figura 2.1).

[8] https://github.com/unrealcv/unrealcv

Tabla 2.1: Comparativa de motores de desarrollo gráfico [4].

Motor Gráfico	Licencia Gratuita	Código Abierto	Market-place	Servicios Adicionales	Lenguajes de desarrollo
UE4 4	Si	Si	Si	Solo de terceros	C++
Panda 3D	Si	Si	No	Compatible VR Pirepheral Net.	C++ y Python
Ogre 3D	Si	Si	No	Compatible Kinect	C++
Quest 3D	Si	No	No	-	C++ y Lua
ShiVa 3D	No	No	Si	Plugin web	Lua
Torque 3D	No	Si	Si	-	C++
Source Engine	Si	Si	No	-	C y C++
Cry Engine	Si	Si	Si	Compatible con Kinect	Python, C#, C++ y Lua
Amazon Lumberyard	Si	Si	No	Amazon Web Services, Twitch integration	C++
Unreal Engine	Si	Si	Si	-	C++
Unity	**Si**	**No**	**Si**	**Unity Ads, Multiplayer, Performance**	**C# y JavaScrip**

2.2 Sistema de simulación multi-cámara MSS

El objetivo principal del sistema es desarrollar un entorno virtual para la simulación de eventos de interés donde se maneje la monitorización con múltiples cámaras dentro del escenario en cuestión, también se puede generar mensajes de difusión desde cada cámara y algoritmos de visión artificial. En la Figura 2.3 se muestra el diseño para un servidor multicliente, y es posible mediante esta arquitectura cliente-servidor conectarse local y remotamente.

Cada módulo permiten una funcionalidad adecuada del sistema, sin embargo existen procesos que se ejecutan secuencialmente, y es donde mediante los hilos se logra una arquitectura paralela para la conexión de múltiples clientes, esta arquitectura permite un flujo adecuado para la transmisión de imágenes [7].

El sistema cuenta con diferentes componentes que trabajan en conjunto para un fin específico, esto implica adaptar; por ejemplo, el procesamiento de imágenes, su procesamientos múltiples, sockets, lógica de sincronización, comunicación TCP y mucho mas, para que funcionen específicamente con el simulador. El diseño modular permite entender de manera mas sencilla la arquitectura del sistema.

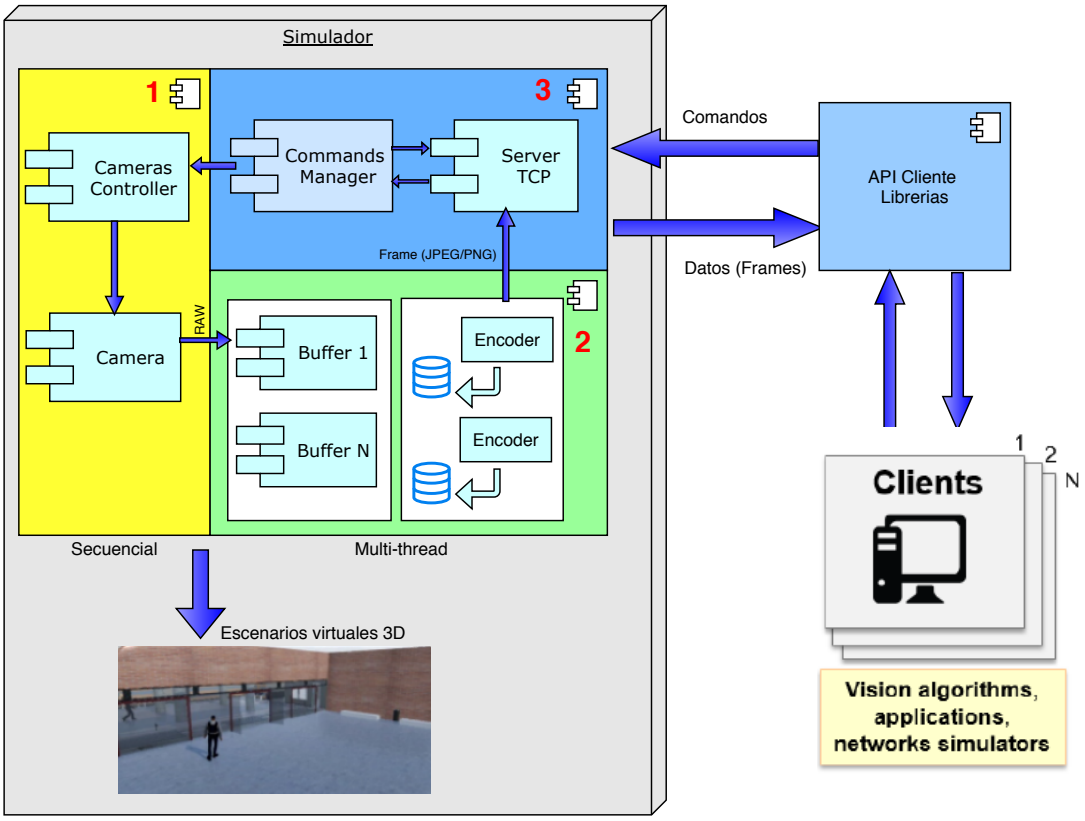


Figura 2.3: Arquitectura cliente-servidor del sistema.

2.2.1 Módulo 1: Gestión Cámaras

El módulo **Gestión Cámaras** (recuadro amarillo) es el encargado de gestionar todo lo relacionado con la integración de Unity, uso del control de cámaras y Unity *Scripting* debido a su lógica de programación orientado a objetos.

Al crear cámaras en el mundo virtual, uno de los factores mas importantes a considerar es la cantidad de *frames* a la que función la cámara, y donde se presentan varios casos; por ejemplo el caso en el que exista una cámara funcionando a 20 fps y otra que se requiera que funcione a 10 fps, es por lo cual se ha considerado los siguiente:

1. Si no hay ninguna cámara, se fuerza a que Unity trabaje a 30 fps.
2. Si hay una sola cámara, el ciclo lógico de funcionamiento modifica a que funcione a los fps que se ha solicitado para dicha cámara.
3. Si hay varias cámaras, se busca el fps máximo que tienen las cámaras y se obliga a Unity a trabajar en ese fps, y las demás en los fps propio para el cual se ejecuto.

El proceso presenta un problema ya que a más imágenes por segundo y a un aumento considerado para crear cámaras, conlleva a un costo computacional grande, por lo que en el caso de que no se pueda generar tantos *frames* en un segundo, los fps de Unity disminuirán y por

consecuencia también los de cada cámara, sin embargo la el sistema no perderá su sincronía y se recomienda trabajar sobre una GPU para que el sistema funcione de la mejor manera.

El simulador incorpora un *script* que permite controlar las cámaras, estas instrucciones provienen del cliente que se conecte al sistema y los comandos desarrollados se muestran en la Figura 2.4 a continuación.

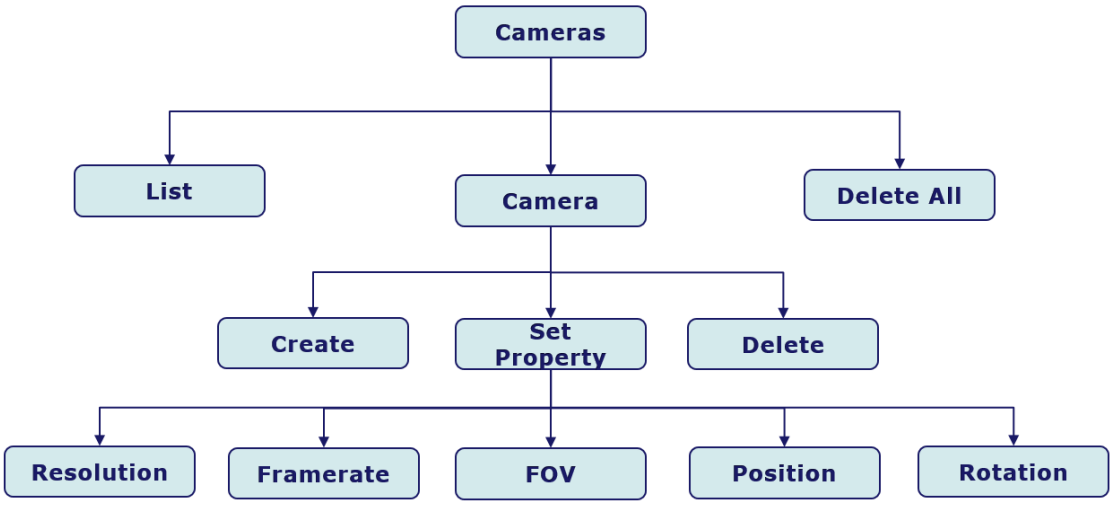


Figura 2.4: Diagrama de funcionalidad de "cameras controller" [7].

2.2.2 Módulo 2: Buffer

El módulo de **Buffer** es el encargado del almacenamiento de *frames* en la memoria temporal y esta implementado en multihilo, para poder trabajar en conversiones de imágenes y su procesamiento (Figura 2.3, recuadro verde).

Este módulo es un hilo que funciona cuando recive un nuevo *frame*, este a su vez llama a la clase "Encoder" la cual hace una conversión al *frame* según la resolución y el formato de configuración (JPG o PNG) de la imagen, el resultado de esta codificación es la que se transmite al cliente, solo es necesario almacenar una imagen ya que trabaja en tiempo real por lo que el buffer es eficiente y los recursos computacionales no son excesivos.

2.2.3 Módulo 3: Servidor

El último módulo es el **Servidor**, el cual es asíncrono y permite conexiones múltiples de clientes. Se encuentra implementado con subprocesos de hilos independientes mediante un *ThreadPool*conectado a los *sockets* del servidor.

Dentro de este módulo se administra los datos recibidos por el servidor y maneja la lista de clientes conectados al simulador en la Figura 2.3 (recuadro azul), se puede observar la lógica de este módulo.

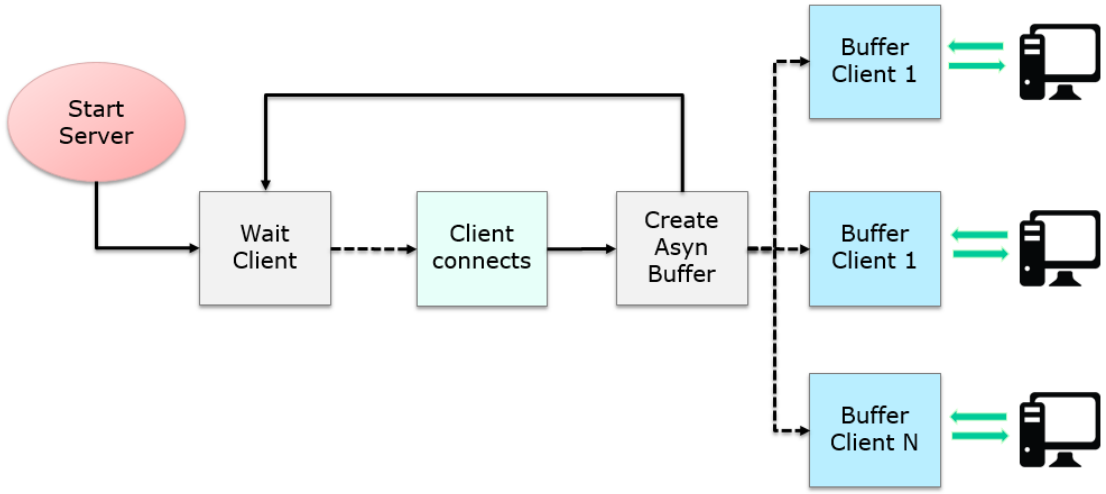


Figura 2.5: Diagrama de servidor TCP [7].

El servidor TCP se implementa con un hilo independiente basado en Microsoft Net[9], desarrollado con socket asincorno, la opción más adecuada para conexión de múltiples clientes, la lógica de esta clase (TCPserverAsyn) se representa en la Figura 2.5.

[9] https://msdn.microsoft.com/es-es/library/fx6588te(v=vs.110).aspx

Capítulo 3

Extensiones del simulador MSS

Este capítulo describe las mejoras realizadas en las librerías del simulador y en la API del cliente. Además se indica distintos modos de ejecución del simulador, el tipo de cámaras existentes y la implementación de una capa semántica en la renderización de vídeo.

3.1 Modos de captura de datos: continuo y bajo demanda

Debido a que los algoritmos de visión artificial, *Deep learning* o conducción autónoma requieren una alta capacidad computacional, esto implica que el tiempo de ejecución de estos algoritmos en casos puntuales tarden más de lo esperado, por lo cual si se ejecutase dentro del simulador cabe la posibilidad de perder el análisis de algún *frame* cuando se transmite vídeo del servidor al cliente, debido a esto el usuario tiene la posibilidad de cambiar el modo de ejecución a tipo *Frame On Demand* (transmitir imágenes bajo demanda de cliente).

La API del simulador MSS incorpora mejoras tanto en el servidor como en el cliente para trabajar en la situación anteriormente descrita, bajo dos modos de ejecución: *Continuous Simulator* y *Frame On Demand Simulator*.

En la Figura 3.1 se describe un diagrama de tiempos cuando se ejecuta el modo continuo (*continuous simulator*), se puede tener dos casos, uno en el que el algoritmo del cliente es lento en analizar los *frames* cuyo caso el servidor sigue generando imágenes más rápido de lo que el cliente puede analizar (timepo $>$ FPS) y se pierde información. El segundo caso en el que el algoritmo procesa de forma rápida (tiempo $<$ FPS) los *frames* y no se pierde ninguna imagen que el servidor envía.

El modo **Continuo** indica que el simulador funciona a la velocidad "normal", es decir, según lo especificado por el usuario al momento de crear cámaras en el escenario o instanciar las que esta previamente en el entorno de prueba.

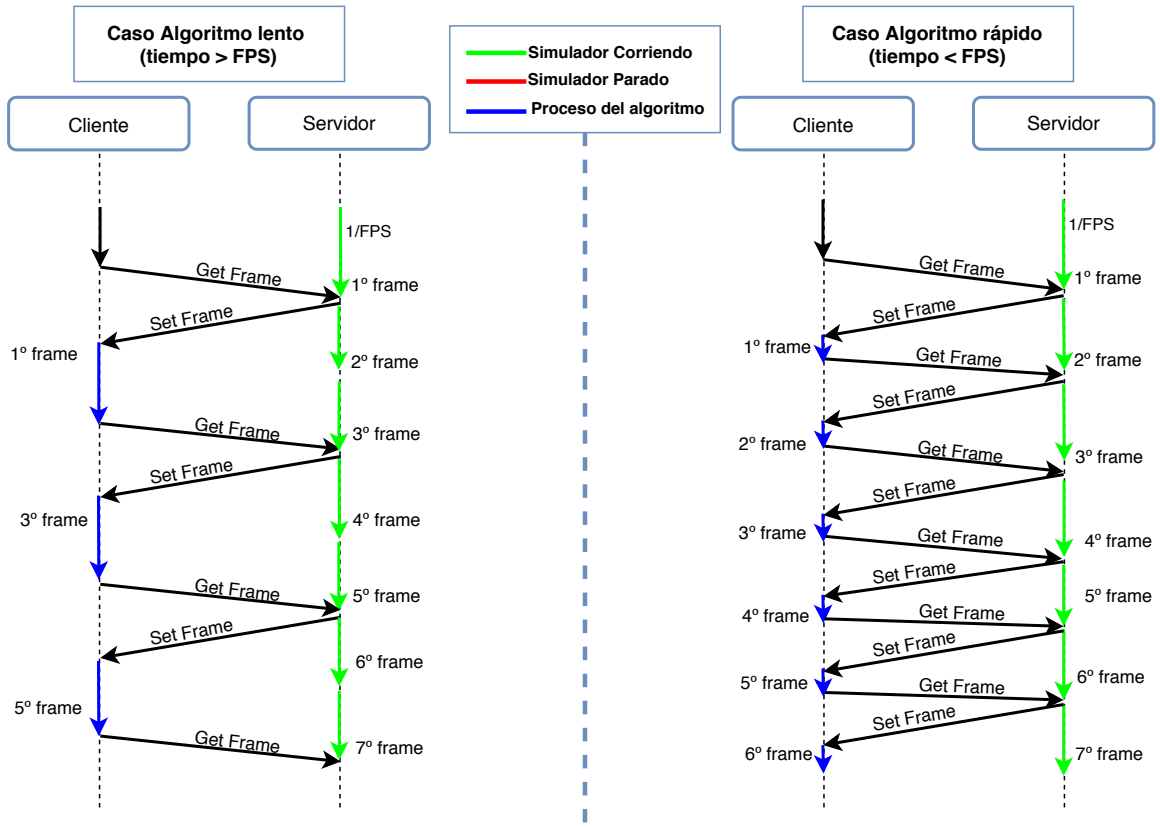


Figura 3.1: Diagrama de tiempos del funcionamiento del modo Continuo (*continuous simulator*) cuando el algoritmo de cliente es lento y rápido

En el caso de usar el modo Bajo demanda no se pierde ningún *frame* ya que en este caso no depende de cuanto tarde el algoritmo del cliente en analizarlos, en este caso el cliente envía al simulador un "*stop*" y después de procesar la información solicita el envío del siguiente *frame*, como se puede observar en el diagrama de tiempo de la Figura 3.2.

Mientras que el modo **Frame Bajo Demanda** permite insertar un tiempo de "pausa" al servidor para retransmitir cada *frame* según el cliente le solicite el envió del siguiente *frame*, de manera que no se pierda ninguna imagen mientras el cliente analiza o procesa algún algoritmo sobre el *frame* que llego previamente. Con tiempos muy pequeños el simulador aparentara ir de forma continua pero a medida que el tiempo aumente se podrá apreciar como el simulador pausa su ejecución el periodo de tiempo solicitado.

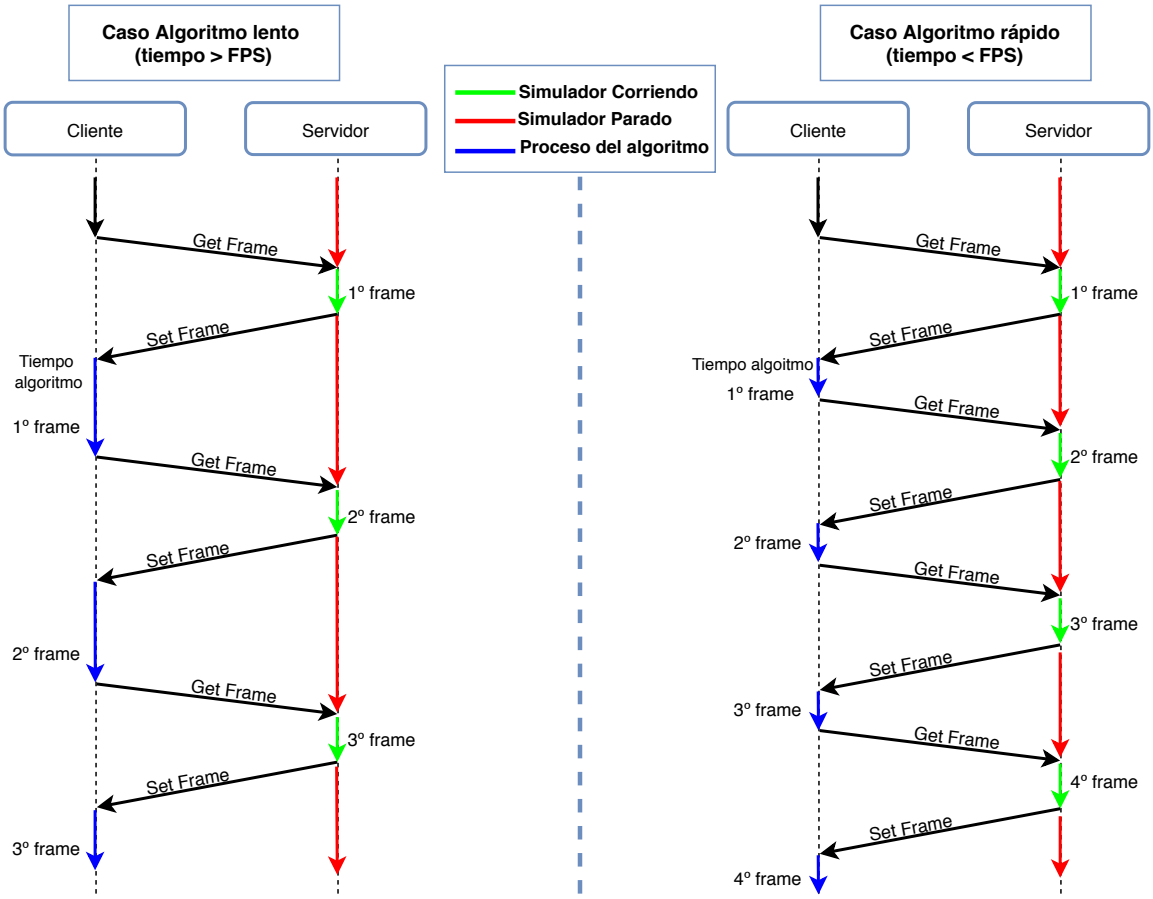


Figura 3.2: Diagrama de tiempos del funcionamiento del modo Bajo Demanda (*Frame on demand simulator*) cuando el algoritmo de cliente es lento y rápido

En la Figura 3.3 se puede observar los métodos correspondientes para activar o desactivar esta funcionalidad desde el cliente al igual que un flujo del funcionamiento básico de estos modos de ejecución. La Figura 3.3a es el *script* programado con un ejemplo para que el simulador funcione en modo "Frame bajo demanda" proporcionando un tiempo de espera de $timeWait = 1/25$, y al finalizar la ejecución se vuelve al modo "continuo".

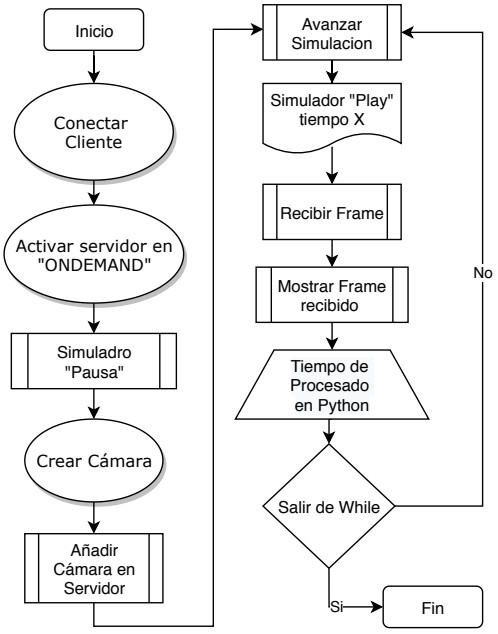
```python
# initialize client
cli = MSSclient()
cli.connectTosimulator(ipAddress, port, sock)
timeWait = 1/10 # tiempo de salto de frame to frame en el simulador (segundos)1/fps
name = "demo2_frameOnDemand"+str(randint(0, 99)) # rand the range 0 to 99;
cam1 = MSScam(name, 640, 480, 10, 0, 10, 0, 10, 123, 0.0)
cam1.addTosimulator(cli.sock, ipAddress, port)
cli.modeFrame(cli.sock, "ONDEMAND") # mode="CONTINUOUS" o "ONDEMAND"
while True:
    vecImgs=[]
    time.sleep(3)  # Simular el tiempo de proceso de algoritmo en python
    cli.advancedSimulation(cli.sock,timeWait)
    ## cam 1
    frame1=cam1.operator()
    if isinstance(frame1,np.ndarray):
        width, height = frame1.shape[:2]
        if width>0 and height>0:
            vecImgs.append(frame1)

    ## generate video wall image
    videowall = dibujarCanvas.makeCanvas(vecImgs, 960, 1)
    ## show image
    if videowall.shape[0]>0 and videowall.shape[1]>0:
        cv2.imshow("Video Wall", videowall) ## display frame
   ## close window & exit preview loop if ESC pressed
    if cv2.waitKey(5) == KEYESCAPE:
        break

# regresar a su estado original la velocidad del Simulador
cam1.removeFromSimulator()
cli.modeFrame(cli.sock, "CONTINUOUS") # simulador funcionamiento continuo
cli.resetSimulator(cli.sock) # rest el simulador

# disconnect from the simulator
cli.disconnectFromSimulator(cli.sock)
```

(a) Código ejemplo de algoritmo



(b) Diagrama de flujo de funcionamiento

Figura 3.3: Código Python del cliente y esquema general de funcionamiento del modo captura "*Frame on demand*"

Mientras en la Figura 3.3b Se observa un diagrama de flujos que muestra los pasos para generar el procesos de la captura de datos en los dos modos descritos. Se puede observar también en el siguiente enlace https://vimeo.com/405353836 el script de ejemplo de la ejecución FRAME ON DEMAND SIMULATOR sobre el simulador y su funcionamiento.

3.2 Cámaras del Servidor

La versión inicial del simulador permite manipular y manejar las cámaras del simulador. En el trabajo previo de MSS [7], se tiene un único objeto para instanciar cámaras desde el cliente. La importancia de incorporar mejoras en MSS se debe a la necesidad de construir nuevos entornos de simulación que permitan entregar un valor añadido a las funcionalidades de las cámaras y probar algoritmos de la visión artificial. Para las mejoras realizadas, el objeto inicial (cámara) se le ha denominado "Cámara de Usuario" debido a que el cliente es quien solicita crear una nueva cámara en el simulador. Con la renovación del módulo en la API de Python se ha incorporado un nuevo objeto del tipo cámara denominado "Cámara de Escena". Éste por su parte tiene la particularidad de ser cámaras ya definidas e instaladas en la escena, únicamente preparadas para inicializarlas y transmitir.

3.2.1 Cámaras: CameraScene y CameraUser

En el funcionamiento del simulador inicialmente un entorno se ejecuta y mediante la conexión de un cliente o varios, se puede insertar una cámara o más en el simulador (*prefab* llamadas **CameraUser**). Además se debe pasar una serie de parámetros a la Clase “Camera” como lo

son; posición, rotación, (ubicación espacial 3D), fps resolución, entre otras. Se puede observar con más detalle en el trabajo MSS [7]. En una versión con nuevos añadidos, es posible predefinir dentro de un escenario 3D de prueba (Servidor) cámaras establecidas en cualquier punto o posición del simulador y acceder a ellas mediante nuevas funciones de la API cliente (Python), estos objetos prefabricados creados en Unity (prefabs) se les ha asignado como **CameraScene**. A continuación se puede observar en la Figura 3.4 cámaras pre-instaladas en la escena (jerarquía de la interfaz de Unity, lineas blancas) y la que el usuario puede crear y posicionar en la escena (azul).

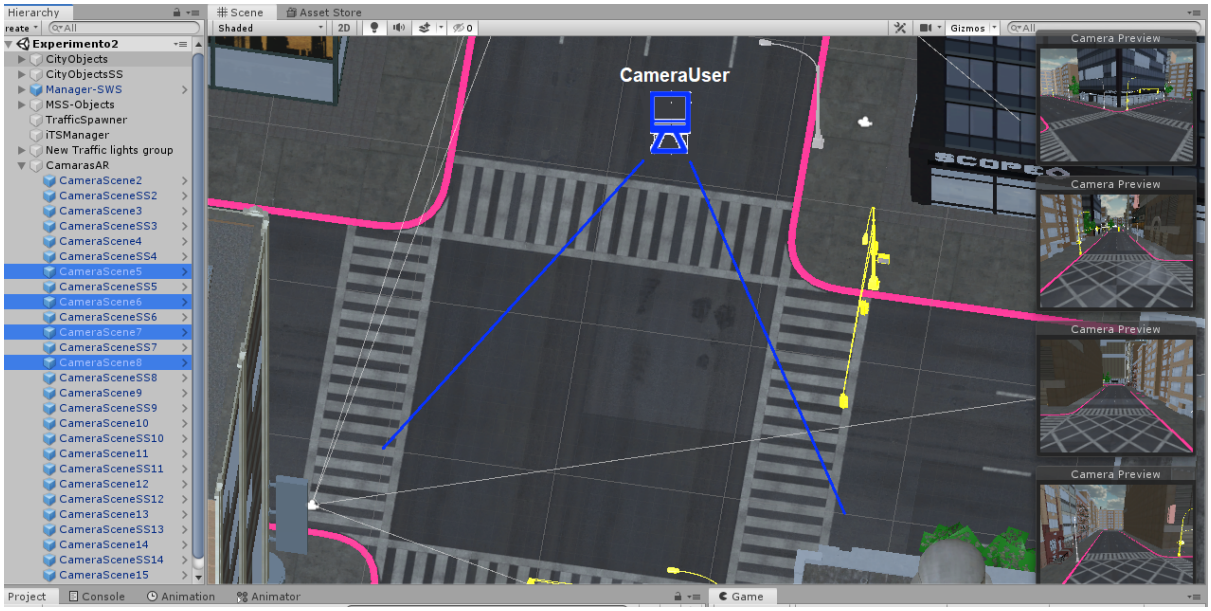


Figura 3.4: Escenario donde se pre-instala el tipo *CameraScene* y donde el usuario puede crear *CameraUser*

3.2.2 Tipo: Fixed Camera, Mobile Camera, PTZ Camera

Los dos prefabs **CameraUser** y **CameraScene** también incorporan una característica adicional en la que se puede definir qué tipo de cámara es la que se quiere insertar, este distintivo se puede asignar independientemente a cada objeto cámara y existen tres tipos: FixedCam, MobileCam y PTZ (observar Figura 3.5)

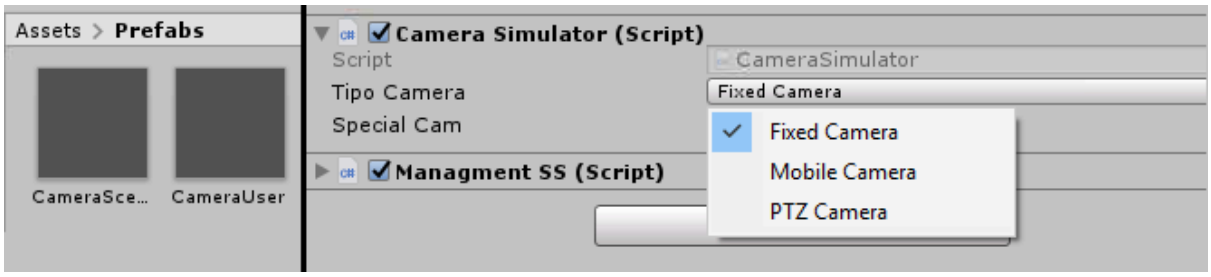


Figura 3.5: Tipo de cámaras posibles para el objeto *CameraScene* y *CameraUser*

Por defecto al insertar una cámara viene definida como *FixedCamera* la cual tiene la car-

acterística de ser una cámara estática y sin la posibilidad de que el usuario pueda realizar ninguna acción sobre ella. Si se selecciona *MobileCamera* esta permite al usuario el control total sobre ella es decir mediante comandos el usuario puede mover o rotar la cámara sobre todo el escenario, y por el contrario una cámara tipo *PTZCamera*, le permite al cliente solo el control sobre su rotación en dos ejes (X, Y) bloqueando por completo la posibilidad de mover a la cámara de posición. Su funcionamiento se lo puede observar en el siguiente enlace https://vimeo.com/405595414. En la Tabla 3.1 se puede observar en detalle las características de esta funcionalidad en las cámaras.

Tabla 3.1: Permisos de funcionalidad en las características de las cámaras

Características	Eje	Fixed Camera	Mobile Camera	PTZ Camera
Posición	X	No	Si	No
	Y	No	Si	No
	Z	No	Si	No
Rotación	X	No	Si	Si
	Y	No	Si	Si
	Z	No	No	No

3.2.3 Cámaras sobre objetos móviles

Existe además la opción (**Special Cam**) para mejorar y ajustar los movimientos de rotación en cámaras que se insertan sobre objetos en movimiento. Actualmente existen cámaras de seguridad ubicadas dentro del transporte publico o privado. Es decir que estos eventos se pueden replicar en el simulador por lo que es posible encontrar cámaras dentro de un autobús, un automóvil o hasta una persona en movimiento. La forma mas adecuada del uso de esta función se puede entender con los siguientes pasos:

- Insertar un objeto que tenga algún tipo de cinemática en la escena (ejem. Helicóptero, Automóvil, Autobús, etc).
- Programar o generar las secuencias de movimiento al objeto insertado.
- Crear un nuevo "GameObject" vació dentro del objeto insertado. Se asigna el nombre "ReferenciaGiro".
- Arrastrar el *prefab* de cámara "CameraScene" dentro del nuevo objeto creado y posicionarlo de la forma que se desee. A está cámara activar la opción ***Special Cam***.
- Ejecutar el simulador y acceder a la cámara para controlar la cámara. En el siguiente enlace se observa más a detalle los pasos para el funcionamiento correcto de esta opción: https://vimeo.com/411703677

Para estos casos las cámaras basan su posición según el objeto “padre” (en la jerarquía de Unity) al que pertenece, ya que dependen del objeto en movimiento. Es decir su posición es relativa a la posición del objeto en movimiento y éste es relativo respecto al mundo 3D de Unity,

para lo cual al activar esta función permite un ajuste en dichos parámetros para que el usuario pueda manipularlo con una mejor precisión. Preferentemente solo se la debe activar cuando una cámara se inserte en GameObject con una desplazamiento en la escena, esta función afecta directamente a cámaras de tipo PTZ ya que el usuario solo tendrá permisos de manipular la rotación de la cámara (no su movimiento o desplazamiento). Para observar esta función se puede observar el siguiente enlace: https://vimeo.com/405602423.

Figura 3.6: Activación de la característica *"Special Cam"* en un helicóptero.

3.2.4 Ventana ejecución principal del simulador

En la incorporación de mejoras del simulador es necesario aumentar el número de variables, métodos y comandos que permitan al cliente enviarlas para que el servidor las pueda ejecutar por lo cual para facilitar el control de estos nuevos parámetros se ha añadido nuevas características en la pantalla de presentación del simulador, como se puede observar en la Figura 3.7, se puede saber a primera vista el número de cámaras que están en escena, diferenciándose del número de cámaras que están en ejecución, es decir, cuantas cámaras están transmitiendo *frames* al cliente, adicionalmente se puede saber de forma permanente el modo de ejecución del simulador.

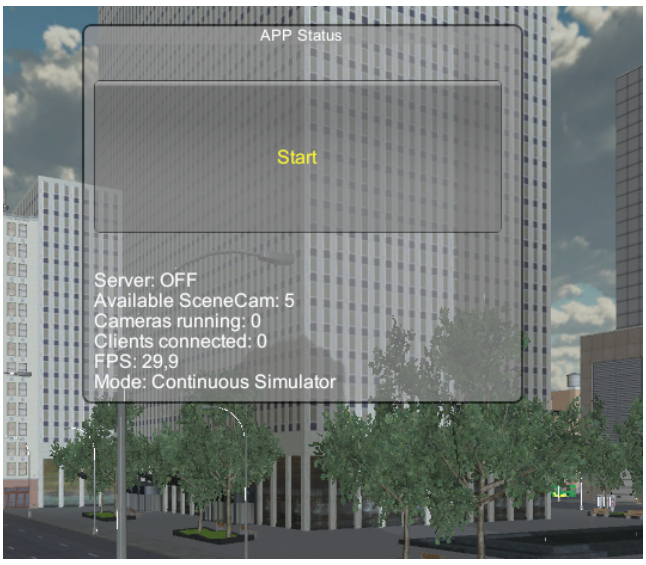


Figura 3.7: Pantalla de presentación del simulador.

A continuación se describe con mayor detalle la información que se presenta en la pantalla del simulador:

1. **Botón Start** .- Este Botón inicializa el simulador, al ejecutar el evento Clic dentro de este se ejecuta funciones para arrancar el servidor y prepara al simulador para la conexión de clientes. Este botón tiene dos estados más asociados: Waiting y Reset.

2. **Server.**– Esta etiqueta muestra el estado de conexión del servidor, es decir que si todos los procesos del servidor arrancaron sin error estará listo para la comunicación cliente-servidor, esta variable va asociada a dos estados: OFF y ON

3. **Availiable SceneCam.-** Esta variable, una vez arrancado el simulador con "Start" muestra en pantalla todas las cámaras que están preestablecidas en la escena, representa los objetos cámara de tipo CameraScene, y lee todas las cámaras que se han colocado tanto fijas como cámaras aleatorias generadas en objetos en movimiento.

4. **Cameras running.-** Esta etiqueta a diferencia de Available SceneCam muestra solo cámaras que se hayan inicializado para transmitir *frames* al cliente, en esta variable se contabilizan tanto las cámaras de Escena (CameraScene) como las cámaras que el cliente vaya insertando (CameraUser) a medida que se conecten más clientes.

5. **Clients connected.-** Esta etiqueta permite visualizar el número de clientes que se encuentren conectados al simulador.

6. **FPS.-** Se muestra la tasa de *frame* (FPS) a la que se esta ejecutando el simulador, es una variable que se obtiene directamente de Unity.

7. **Mode.-** Muestra el modo de ejecución en el que se encuentra el simulador, descrito en el apartado 3.1, este parámetro está asociado dos estados: *Continuos Simulator* y *Frame On Demand Simulator*.

3.3 Mapa semántico

Esta sección se describe brevemente lo que trata la segmentación semántica dentro de la visión artificial y el protocolo llevado a cabo para su implementación dentro del simulador.

3.3.1 Segmentación semántica

El principal objetivo del simulador es crear un sistema multi-cámara distribuido para la transmisión de vídeo (conjunto de imágenes a determinados FPS) en una red cliente-servidor, donde el desarrollo de las simulaciones se lleva en espacios exteriores (Escenario Ciudad), esto permite segmentar de manera concreta el "cielo", "suelo", "aceras", "arboles", "automóviles", "personas", etc, lo que se considera el conjunto de clases necesarias para la segmentación semántica (ver Figura 3.8).

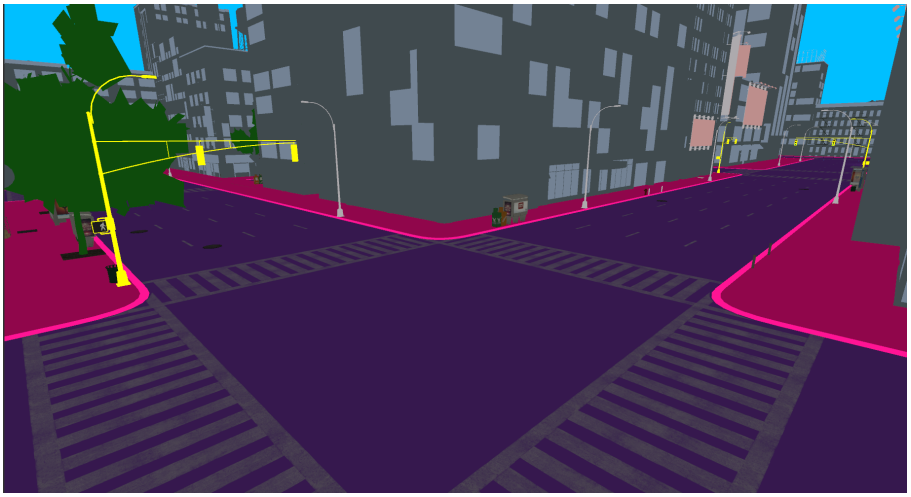


Figura 3.8: Aplicaciones de segmentación semántica sobre el simulador MSS

Para seleccionar el modelo se segmentación es importante considerar una red que haya sido entrenada con un *dataset* que tenga las clases características del simulador, debido a lo cual el protocolo de clasificación se basa (solo como referencia) en los artículos relacionados con la base de datos *CityScapes* [20]. Esta base de datos es una de las más extendida para ambientes en exteriores (ciudades), por lo que sirve como base para muchos proyectos enfocados a la visión artificial, para el simulador MSS se ha tomado como referencia el trabajo [21], del cual se puede tener una visión general de las clases y asignación de colores para la clasificación de objetos.

- **Seleccionar número de clases**

Tabla 3.2: Clasificación por clases y asignación de etiquetas por colores

Clase		Color[1]	RGB Unity	Hexadecimal
Road	Carretera	Rebeccapurple	(102,51,153)	#663399
Sidewalk	Acera	Deeppink	(255,20,147)	#FF1493
Buildings	Edificios	Lightslategray	(119,136,153)	#778899
Buildings (complements)	Ventanas, Puertas	Slategray	(112,128,144)	#708090
Billboards	Vallas publicitarias	Rosybrown	(188,143,143)	#BC8F8F
Pole	Poste (Estructura)	Darkgray	(169,169,169)	#A9A9A9
Traffic_light	Semáforo	Yellow	(255,255,0)	#FFFF00
Vegetation	Vegetación	ForestGreen	(34,139,34)	#228B22
Sky	Cielo	Deepskyblue	(0,191,255)	#00BFFF
Person	Personas, peatones	Red	(255,0,0)	#FF0000
Car	Automóviles	Navy	(0,0,128)	#000080
Bus	Autobuses	Teal	(0,128,128)	#008080

Las etiquetas que se observa en la Tabla 3.2 se basa en el trabajo [20], sin embargo las clases que se observan son autoría propia del TFM ya que se guía por los objetos existentes dentro del proyecto del simulador.

Para generar mapas semánticos en las escenas del simulador, se consideran un total de 12 clases, estas pueden añadirse según las necesidades y exigencias que requieran nuevas escenas, es decir que con otras simulaciones donde se incorpore nuevos objetos (o más clases por consecuencia un nuevo color) como bicicletas, trenes, aviones, etc. se puede sumar a las prestaciones del simulador. La Tabla 3.2 presenta las etiquetas seleccionadas con las que se conforman las distintas clases y las cuales se implementan dentro del entorno del servidor en Unity, con esto se obtiene el *ground-truth* de forma precisa para poder ser comparado con los algoritmos de terceras personas.

- **Crear materiales y/o texturas**

Una vez identificado el número de clases existen en el simulador, el paso a seguir es crear los distintos materiales y/o texturas de los colores que indiquen el etiquetado para asignar a cada objeto de la escena, como se observa en la siguiente Figura 3.9 los colores asignados son igual a la Tabla 3.2, donde a consideración del diseñador se crea un material que define acerca de cómo la superficie debería ser renderizada o por otro lado una textura que sirven para imágenes bitmap, para mayor detalle se puede ver la documentación de Unity en https://docs.unity3d.com/es/530/Manual/Shaders.html

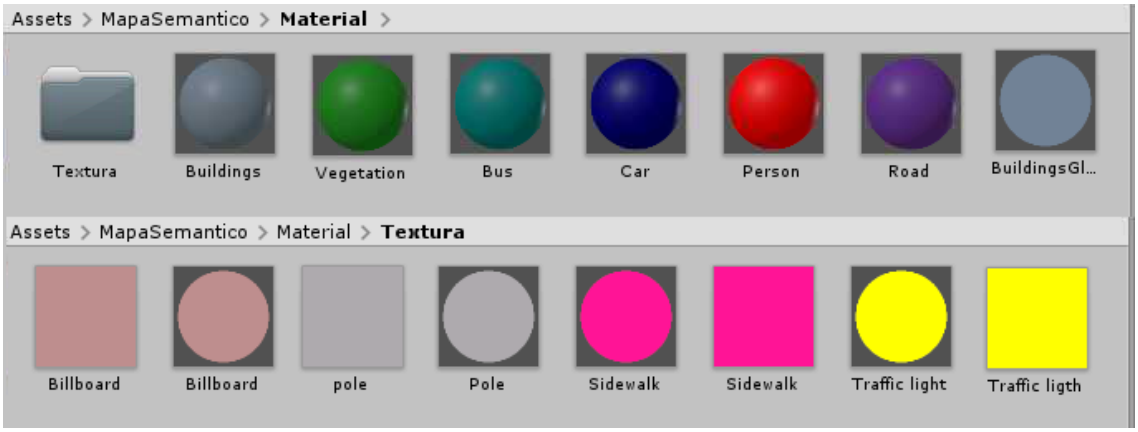


Figura 3.9: Materiales y texturas para etiquetar cada objeto en los escenario.

Para el etiquetado del cielo (clase *sky* o *background*) es necesario manipular el parámetro ***Clear Flag*** en el componente ***Camera*** del objeto **Cámara** para seleccionar: *skybox* (cielo real) o *solid Color* (asignar un color de fondo), el resultado de esto se puede observar en la Figura 3.10.

[1] https://htmlcolorcodes.com/es/nombres-de-los-colores/

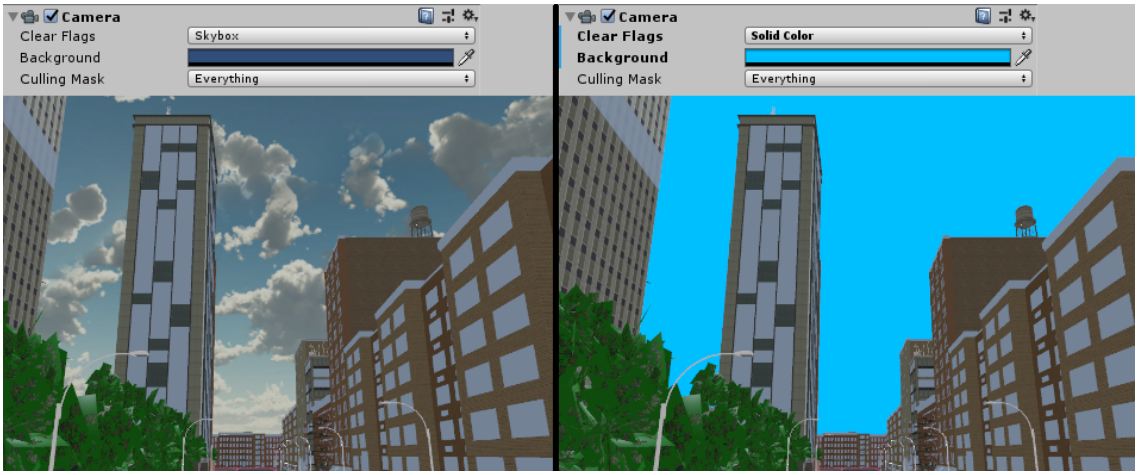


Figura 3.10: Asignación de etiqueta "*sky*" para la renderización RGB o mapa semántico.

- **Creación de capas**

El uso de capas en Unity permite a las cámara renderizar solo una parte de la escena por lo cual se puede indicar selectivamente que objetos pueden o no mirar por una cámara. Para lo cual se crean dos nuevas capas: ***MainStage*** (renderización RGB) y ***SemanticStage*** (renderización Mapa semántico).

Para que una cámara pueda renderizar una parte o ciertos objetos de la escena es necesario el uso de **Capas** (*Layers*), en el entorno de la ciudad se añadieron dos capas nuevas, como se puede ver en la Figura 3.11, la primera ***MainStage*** en la cual todos los objetos tiene su propio material y textura tal cual se ha desarrollado todos los ejemplos de simulación (RGB), y la segunda ***SemanticStage*** una capa personalizada donde cada objeto de la escena esta clasificado con una etiqueta (color) siguiendo el protocolo de la Tabla 3.2.

Figura 3.11: Creación de capas para renderización real (RGB) y mapa semántico

- **Asignación de capa y material a los objetos**

Una vez creado los materiales y las capas se debe proceder a la asignación de los mismos, esto quiere decir que se establecerá cada material para etiquetar los objetos y una de las dos capas para indicar a las cámaras la renderización de ciertos objetos en una escena. Como se

observa en la Figura 3.12 se debe seleccionar el objeto y en el inspector en el parámetro *Layer* asignar la capa correspondiente al mismo.

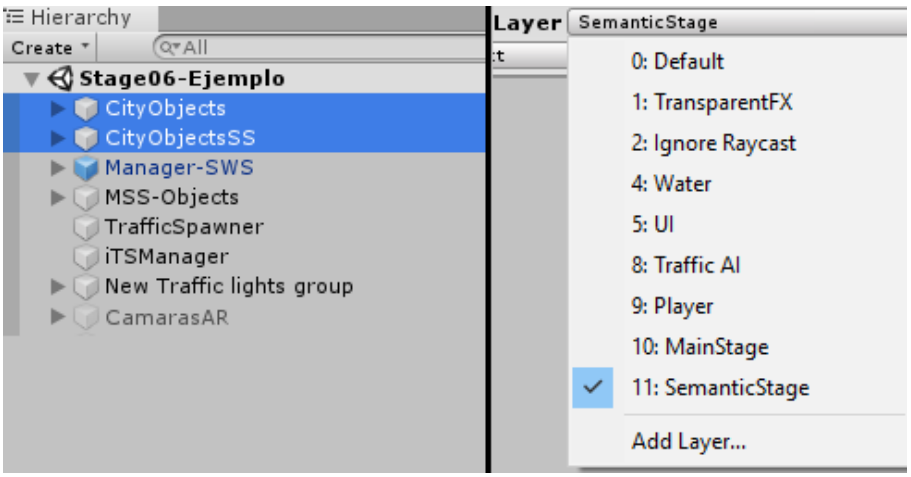


Figura 3.12: Asignación de determinada capa a uno o varios de los objetos de la escena.

En el caso de asignar el material es necesario seleccionar el objeto en cuestión (en el ejemplo edificio), dirigirse a su componente *Mesh Collider* y arrastrar el material determinado para asignar a la clase correspondiente, para obtener un resultado final como se puede ver en la Figura 3.13.

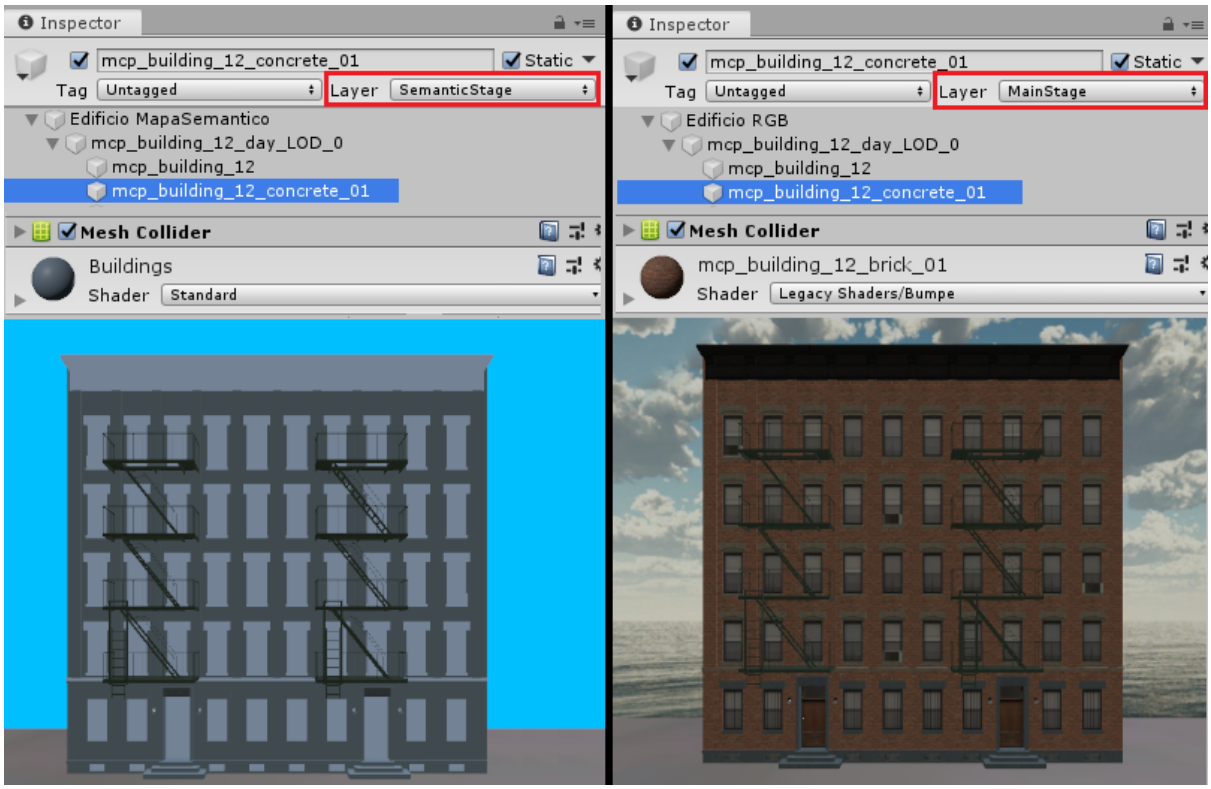


Figura 3.13: Asignación del material y/o textura según su correspondiente clase (edificio).

- **Crear una escena con etiquetado y mapa semántico**

  Al momento de implementar un mapa semántico en las simulaciones, el procedimiento que se siguió fue el siguiente:

  1. Identificar los objetos del escenario "*CITY*" dentro de *Hierarchy* de Unity para poder clasificarlos según su clase.
  2. Duplicar (ver Figura 3.14) todos los objetos existentes que aparecerán en la escena del simulador. De esta forma se tendrá repetidos dos veces todos los objetos existentes, así el uno servirá para tener una escena "normal" (RGB) y la otra para etiquetar con colores y tener el mapa semántico (*ground-truth*). Hay que tomar en cuenta que para un mejor rendimiento del simulador se recomienda tener la menor cantidad de objetos posibles en una escena, por lo que se puede buscar alternativas a este método para incorporar una capa semántica.
  3. Asignar a cada objeto el material correspondiente según su clase (persona, bus, edificio, vegetación, etc)
  4. Asignar a cada objeto duplicado la capa de *SemanticStage*.
  5. Asignar a los objetos principales la capa de *MainStage*, estos objetos tendrán las texturas y materiales originales, es decir los colores RGB.

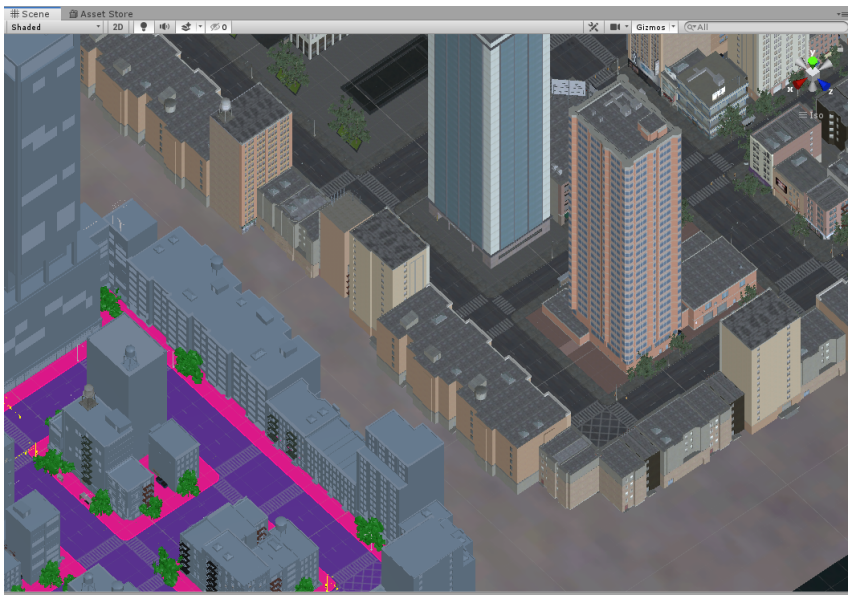
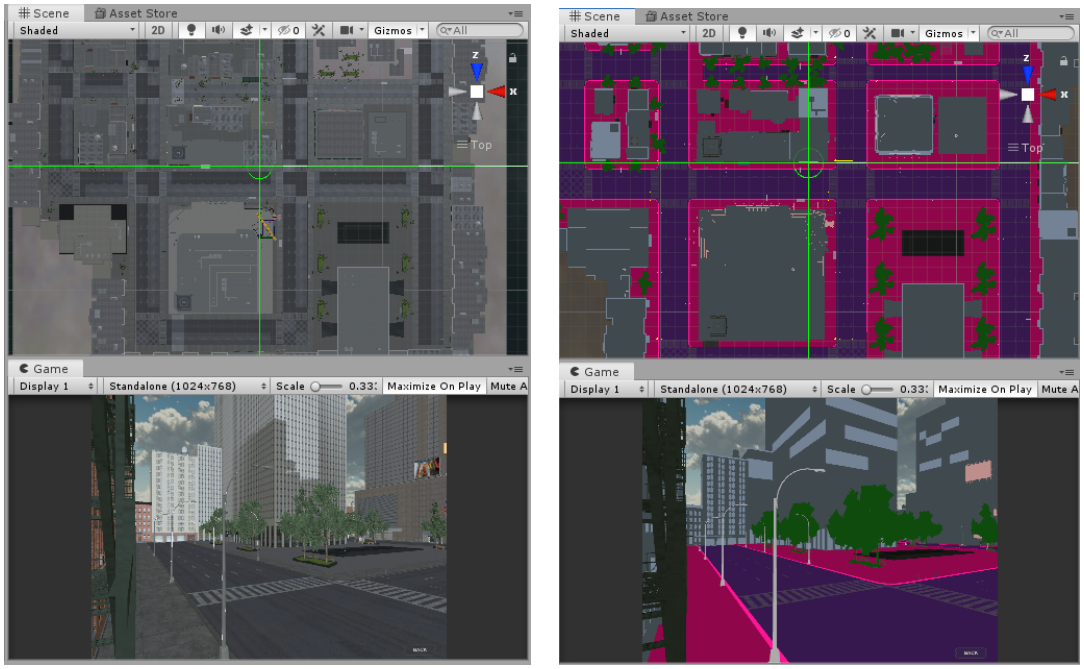


Figura 3.14: Duplicado de todos los objetos (ciudad) para asignar capas distintas una RGB y otra mapa semántico para etiquetar cada objeto según su clase.

La finalidad de duplicar y asignar capas a los objetos de la escena es que al instanciar una cámara en el simulador ésta pueda renderizar los objetos sobre ella según su capa, es decir que si se indica a la cámara que visualice la capa *MainStage* solo se gráfica dichos objetos asignados a esa capa, mientras que si se selecciona *SemanticStage* la cámara solo proyectara los objetos asignados a esa determinada capa. La Figura 3.15 muestra como debe quedar los objetos RGB y los objetos etiquetados por color.

Figura 3.15: Ejemplo de objetos duplicados para asignarlos a distintas capas y que las cámaras puedan renderizar la capa RGB o Mapa semántico.

Un ejemplo de los escenarios finales se puede observar a continuación, en la Figura 3.16a se observa el escenario "normal" o RGB, mientras que para la capa de mapa semántico todos los objetos están clasificados con las distintas etiquetas y la escena que observa la cámara es similar a la que se puede ver en la Figura 3.16b.



(a) Escenario con capa *MainStage* (b) Escenario con capa *SemanticStage*

Figura 3.16: Presentación de escenarios con capas de renderización y etiquetado de objetos

- **Gestión de Visualización y Renderización**

Después de asignar las capas y el etiquetado de los objetos es necesario gestionar cuando observar los objetos en RGB y cuando visualizar los objetos clasificados por color, por lo cual dicha gestión se hace con el *script* ***Managment SS***. Este código se ha desarrollado para añadir a las cámaras ***CameraUser*** y ***CameraScene*** y controlar si la cámara renderiza los objetos con la capa *MainStage* (objetos RGB) o los objetos con la capa *SemanticStage* (mapa semántico), como se observa en la Figura 3.17 este componente permite seleccionar si se activa o desactiva determinada capa mediante la opción ***Semantic Segmentation***.

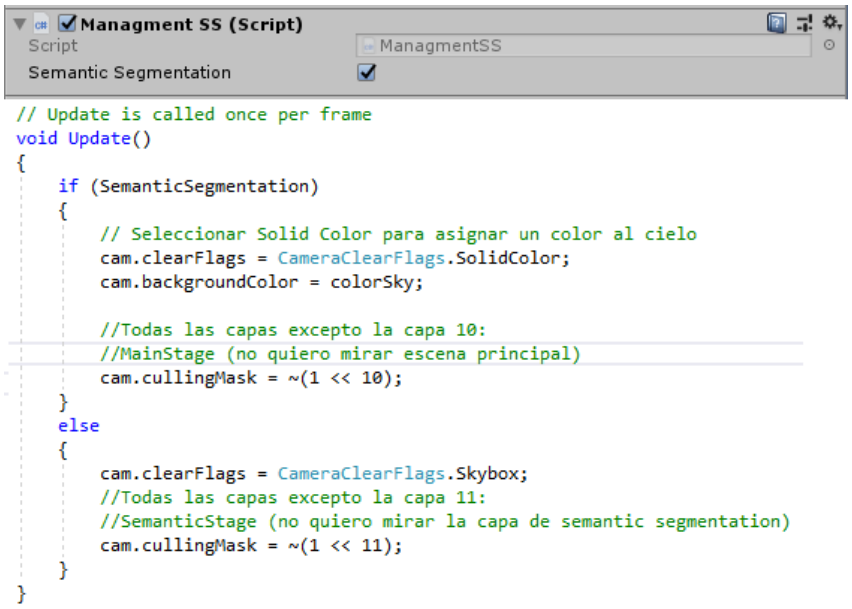


Figura 3.17: Código para activar o desactivar mapa semántico en el renderizado de las cámaras del simulador

Para gestionar que capa renderiza la cámara se hace uso de la variable *Culling Mask*, la cual permite personalizar la forma en que se renderiza los fotogramas al instanciar una cámara en la escena. Para entender de forma sencilla la creación de una escena completa paso a paso se puede observar el vídeo a continuación: https://vimeo.com/416959724.

En el lado del cliente, el *script* **prueba1_MapaSemantico.py** de la API de Python es un ejemplo el cual crea un objeto de tipo ***CameraUser*** para instanciar nuevas cámara en la escena y a su vez activarla con la opción *Semantic Segmentation*. Como se observa en la Figura 3.18.

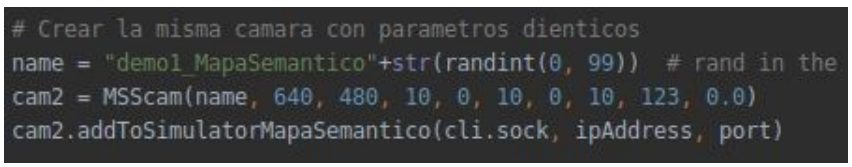


Figura 3.18: Cámara semántica activada desde el cliente.

*__Build de escena.-__En el siguiente enlace se puede observar con detalle la configuración al construir una escena en Unity (*Build*): https://vimeo.com/453247543

Capítulo 4

Desarrollo de API en lenguaje Python para el simulador MSS

Esta sección describe la creación y ejecución de la API de Python en la versión V 3.6.9, desarrollada para funcionar como cliente en el simulador MSS, se presenta y describe la estructura de la API al igual que las clases y funciones que permiten su funcionamiento con distintos test de ejemplos. Se detallan los puntos más importantes para preparar un entorno dentro del sistema operativo Linux o en este caso en concreto en una máquina virtual de Linux/Ubuntu (V. 18.04.4 LTS), sin embargo se puede usar en otras plataformas.

4.1 Funciones de la API

El desarrollo de una API permite la facilidad de comunicación entre aplicaciones, en éste caso se suministra bibliotecas en el cliente de Python para interactuar con los escenarios del simulador. La API permite tener un código muy reducido para el desarrollo de diferentes aplicaciones en el cliente. La API programada en el lenguaje Python trae ventajas competitivas en la programación de alto nivel, ya que es ampliamente utilizada en la actualidad para investigación de la visión artificial. Dentro de el desarrollo de la API se encuentras las siguientes librerías:

- **MSScam.py.-** Mediante esta librería se define la estructura de las cámaras a través de la cual se manejan y controlan instancias del objeto cámara del simulador. Dentro se encuentran distintas funciones que permiten, del lado del cliente, manipular parámetros respecto a las propiedades de la cámara. El constructor de esta clase permite instanciar una cámara genérica en el simulador, además de que el usuario puede inicializar el objeto según las necesidades. Este archivo es el que más funciones posee, ya que controla aspectos como: asignar ID a cada cámara, registrar el objeto en el simulador, manipular la resolución y codificación de imagen, transmisión de *frames*, movimientos de posición y rotación, etc.
- **MSSclient.py.-** Esta clase contiene todos los métodos que se relacionan con la comunicación y creación entre clientes y el simulador. La comunicación que se establece es a través de *sockets*. La librería permite a la API gestionar parámetros relacionados con el simulador, por ejemplo: establecer la conexión, remover el cliente del simulador, gestionar, gestión en la velocidad de la simulación para transmisión de *frames*, número de cámaras que el cliente ha creado en el simulador, etc.
- **MSScam_control.py.-** Esta librería permite gestionar una interfaz gráfica de usuario para mostrar los *frames* recibidos de una cámara y manejarla en tiempo real. Esta interfaz muestra opciones de usuario para manipular la posición y rotación de la cámara, es decir a través de botones de control se envían comandos conjuntamente con las demás clases para actuar sobre una cámara (Ejemplo en la Figura 4.5).

- **MSScam_insert.py.-** Con la API, ésta clase permite instanciar y crear una "cámara de usuario" [1] mediante una interfaz gráfica de usuario. Es decir que se puede insertar un objeto cámara en alguna posición 3D del escenario de simulación de forma gráfica.

La implementación de la API permite generar varios ejemplos mediante el llamado a los métodos de las libarías incorporadas. Esto permite desarrollar ejemplos con un código mas genérico y sencillo de implementar, en la Figura 4.1 se puede observar un diagrama de bloques general de la implementación y funcionamiento de la API en Python y parte de las funciones de las librerías que contiene.

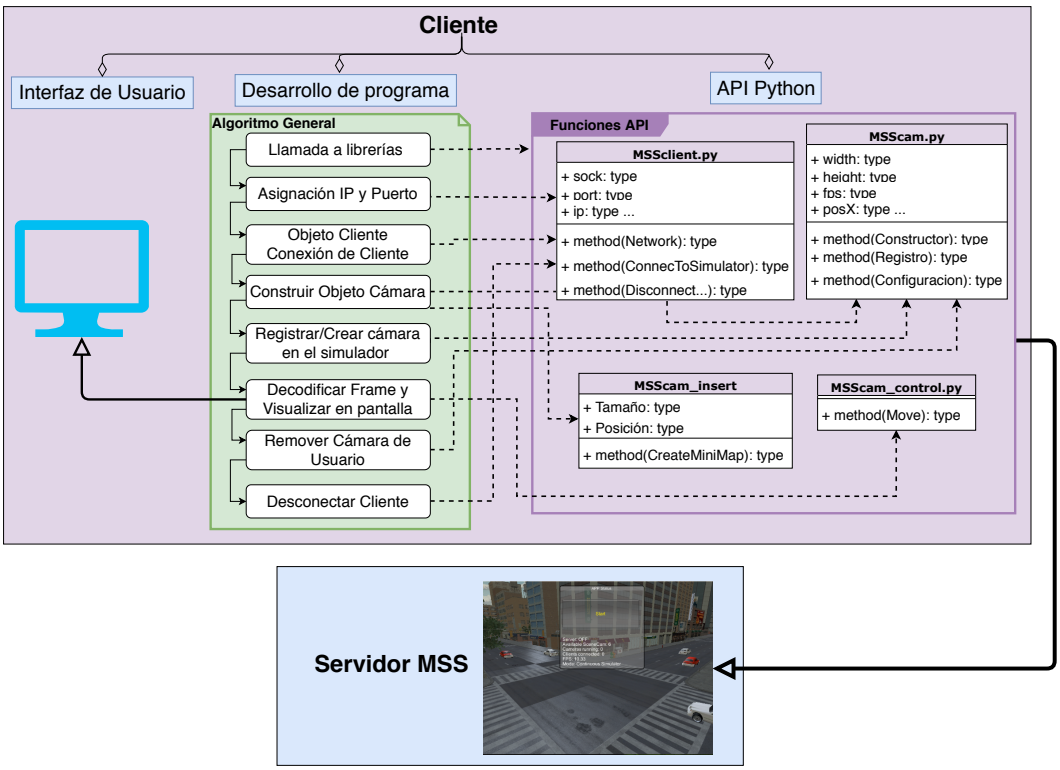


Figura 4.1: Diagrama de bloques de los módulos implementados.

4.2 Ejemplos de ejecución

Los módulos implementados en la API de Python junto con las mejoras del servidor para generar nuevos escenarios, permiten desarrollar varios ejemplos para conocer el uso y aplicación de las funcionalidades de la API. Después de preparar el entorno[2], la ejecución de la API de Python es muy sencilla, dentro de la carpeta de entorno (env) se crea un nuevo directorio "CodesPython" donde se encuentra los ficheros ejemplos para la ejecución.

En la Tabla 4.1 a continuación se muestra las verificación de los ejemplos ejecutados con las funcionalidades implementadas y verificadas. Después de migrar la API cliente al lenguaje Python, el simulador parte con 6 test de ejemplos, estos se realizan a través del usuario directamente creando cámaras en el simulador (***CameraUser***). Posteriormente, las mejoras implementadas permiten definir previamente cámaras dentro del escenario (***CameraScene***) y se muestra el modo de ejecución.

[1] En la sección 3.2 se explica ampliamente las "cámaras de usuario" y "cámaras de escena".
[2] Consulte el Anexo A para obtener más detalle sobre configuración de cliente en Python

Tabla 4.1: Resumen de ejemplos de código funcionales

Script	Funcionalidad	Tipo Cámara	Observación
testbasic.py	addTosimulator	CameraUser	Insertar una cámara con características básicas
testbasic2.py	saveToFile	CameraUser	Insertar una cámara desde un archivo de texto
testbasic3.py	setFPS setTXRXformat	CameraUser	Cambiar FPS y calidad de imagen de las cámaras
testcamcontrol.py	GUIcontrol	CameraUser	Interfaz de usuario para controlar movimientos de la cámara
testcaminsert.py	GUIinsert	CameraUser	insertar cámara a partir de interfaz gráfica
testvideowall.py	operator	CameraUser	Generar una display con múltiples cámaras
ejem1_RAcamera.py	readTosimulator	CameraScene	Identificar número de cámaras puestas en la escena
ejem2_videowall.py	configRACamera	CameraScene	Seleccionar por nombre e ID las cámaras de la escena y visualizar en una pantalla de varias cámaras
ejem3_videowallSelect.py	setIDCamera	CameraScene	Seleccionar las cámaras que el usuario desea visualizar en una pantalla de displays
ejem4_ReadRemove.py	removeCameraSceneSimulator	CameraScene	Eliminar cámaras de escenario pre-establecidas en el simulador
ejem5_RACamControl.py	GUIControlRACam	CameraScene	Controlar movimiento y posición de cámaras de escenario
ejem6_initRACamera.py ejem6_viewRACamera.py	readTosimulator configRACamera	CameraScene	Inicializar cámaras del escenario y acceder con varios clientes para visualizar
prueba1_MapaSemantico.py	addToSimulatorMapaSemantico	CameraUser CameraScene	Insertar cámaras de usuario con mapa semántico o RGB
prueba2_FrameOnDemand.py	modeFrame	CameraUser CameraScene	Modificar la velocidad de simulador para transmitir frames bajo demanda del usuario.
prueba3_CaptureVideo.py	write(frame)	CameraUser CameraScene	Almacenar frames de las cámaras para almacenarlos en vídeo

4.2.1 *Scripts* Cámaras de Usuario

Como se puede observar en la Figura 4.2 se ingresa el parámetro de la ip del servidor junto con el comando de ejecución de un archivo Python:

```
(env)~$ Python testbasic.py -ip XXX.XXX.XXX.XXX -port YYYY
```

Por defecto si no se envía ningún parámetro al archivo de ejecución *testbasic.py* el fichero CLParser proporciona la IP de localhost 127.0.0.1 y el puerto 8889 si el servidor en el mismo ordenador y misma red, sin embargo se puede trabajar sobre este archivo para simplificar las ordenes de ejecución (dentro del directorio Utils).

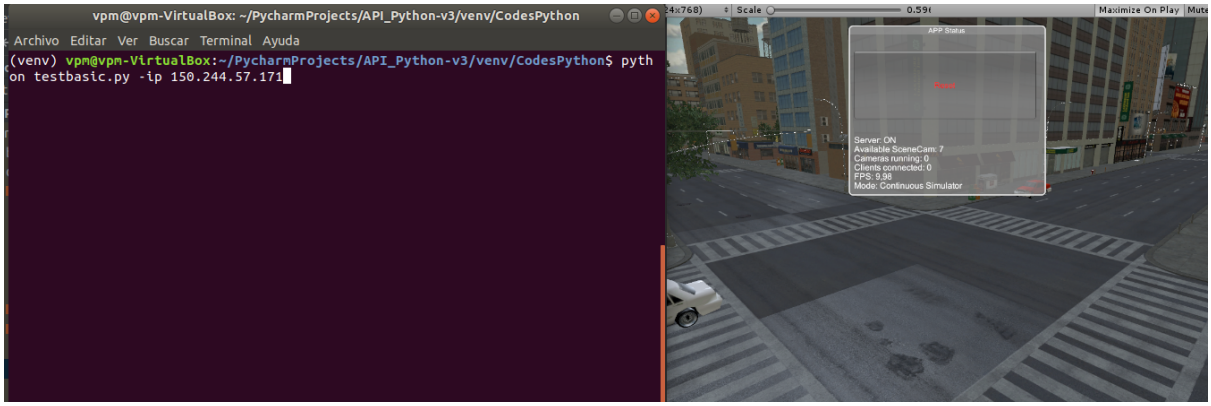


Figura 4.2: La ejecución de testbasic.py se crea una cámara y se visualiza en la API

Para el segundo ejemplo el objetivo es leer de un archivo de texto los parámetros para instanciar una cámara, este fichero debe tener la siguiente estructura:

```
//   each line contains a camera with the following format:
//   name/fps/width/height/TXRXformat/JPEGquality/positionX/
//   positionY/positionZ/rotationX/rotationY/rotationZ
//   example: camtest/10/640/480/0/75/9.816450/3.844200/15.368404/
//            0.000000/0.000000/0.000000
demo2_test/10/640/480/0/75/10.0/-4.0/7.0/20.0/10.0/0.0
```

En la Figura 4.3 se crea una cámara nueva en el simulador la cual, todos sus parámetros (resolución, posición, orientación, etc) se guarda en un archivo de texto y posteriormente es leída de dicho fichero para instanciar una nueva cámara con dichas características, se puede modificar los parámetros si se desea.

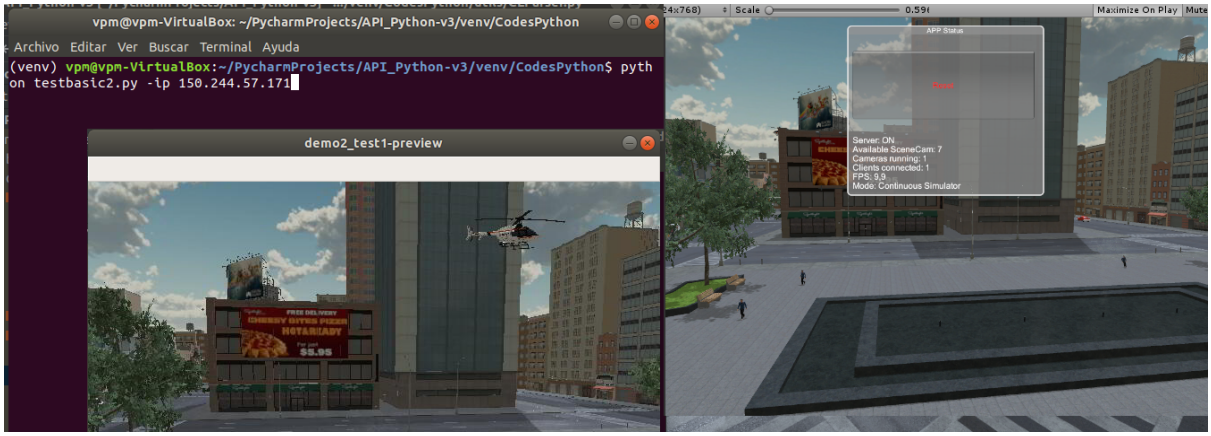


Figura 4.3: testbasic2.py permite guardar y leer desde un archivo de texto la configuración de una nueva cámara.

La forma de ejecutar el ejemplo 3 de la API se puede observar en la Figura 4.4 la cual, después de crear una cámara dentro del simulador, inserta diferentes calidades de resoluciones a la pantalla de visualización del cliente.

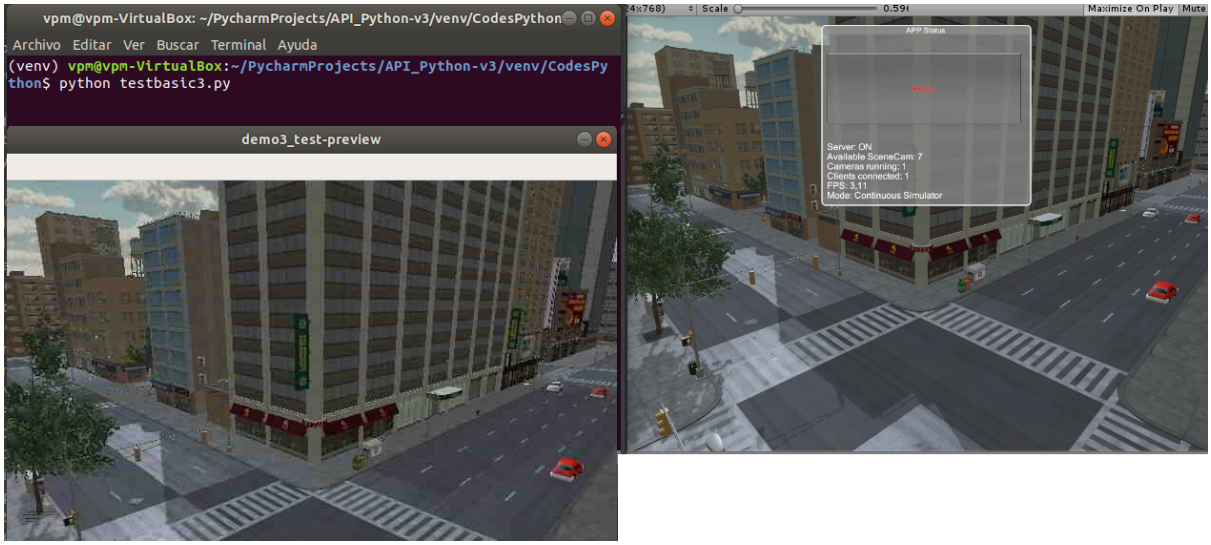


Figura 4.4: La ejecución del fichero testbasic3.py permite cambiar el formato de la imagen en JPG o PNG.

EL fichero de Python *testcamcontrol.py* es un ejemplo en el que permite trabajar con interfaz de usuario, es decir que en tiempo de ejecución del cliente se puede variar diferentes parámetros de la cámara, por lo tanto se puede controlar la posición espacial de la cámara, la rotación, y la orientación de dicho objeto, la interfaz dentro del ejemplo se observa en la Figura 4.5.

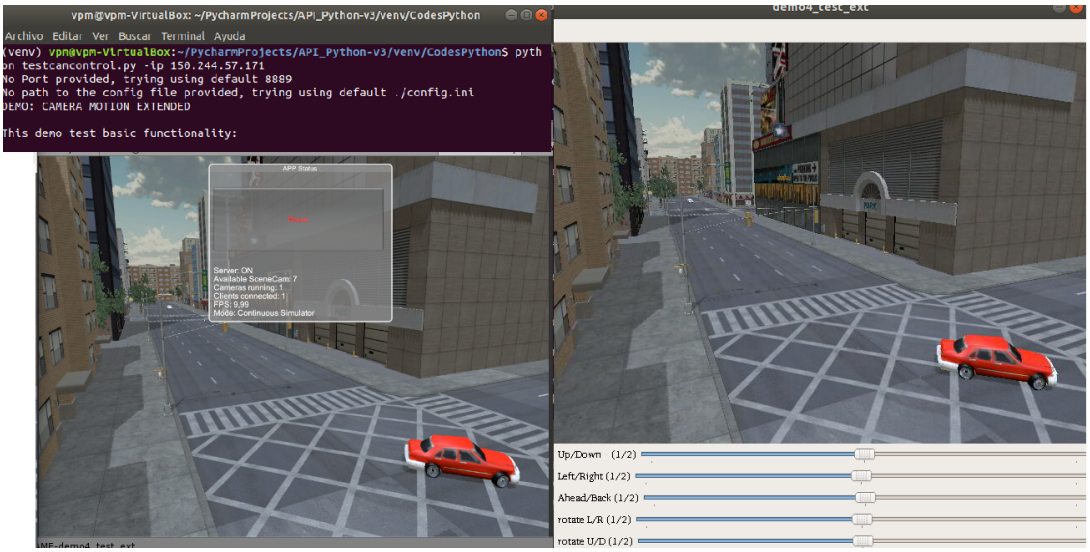


Figura 4.5: El fichero testcamcontrol.py crea una cámara con una GUI para controlar diferentes parámetros de la cámara.

testcaminsert.py es un archivo que permite al usuario, de forma intuitiva y gráfica, ubicar un cámara dentro del entorno del servidor, el cliente puede insertar dicho objeto en cualquier posición y orientada en cualquier dirección que se desee, en primera instancia se muestra el plano "XY" (vista de pájaro) y después de seleccionar la ubicación pasa a una segunda instancia donde se selecciona la altura de la cámara (plano Z), ver Figura 4.6.

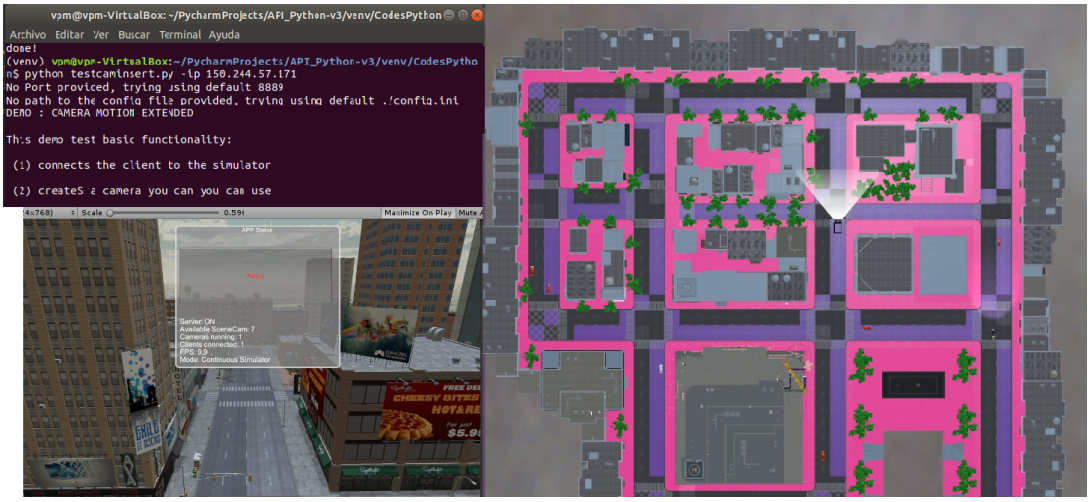


Figura 4.6: Con el ejemplo testcaminsert.py es posible insertar una cámara mediante una GUI

Este último ejemplo crea un cliente con 6 cámaras ubicadas en distintas posiciones, para posteriormente mostrarlas en una sola pantalla como se pude observar en la Figura 4.7. Se puede observar la funcionalidad de la API de Python en el siguiente enlace: https://vimeo.com/365731811.

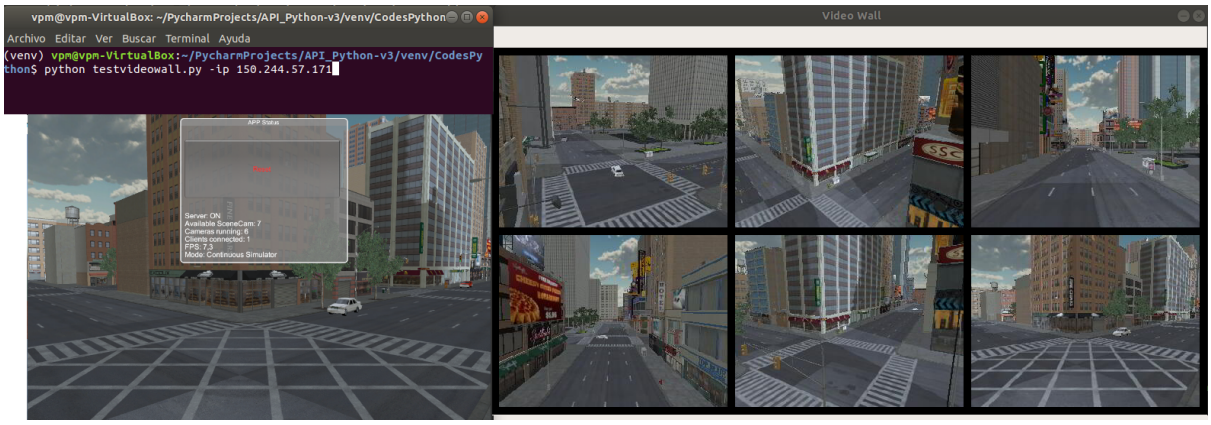


Figura 4.7: La ejecución del fichero testvideowall.py permite visualizar en un panel distintas cámaras del simulador

4.2.2 *Scripts* Cámaras Escenario

Parte de las mejoras desarrollados en la API del cliente y del servidor es instalar previamente cámaras en un escenario, este tipo de cámaras se explica con mayor detalle en la sección 3.2. Las funciones del cliente permite inicializar dichas cámaras para empezar a transmitir por lo que se ha determinado asignarlas como cámaras del escenario.

Las cámaras de escenario son parte de las librerías de la API de Python por lo cual es importante tener presente que el servidor debe estar actualizada para soportar la nuevas funciones en Python, la nueva versión del simulador MSS se explica en el capítulo 3. Para el uso de los ejemplos de cámaras de escenario se ha desarrollado algunos *scripts* con diferentes modelos mostrando el uso de los nuevos métodos de la API, estos archivos como se puede observar en la Figura 4.8 son todos los que empiezan por **ejem**X_demostracion.py

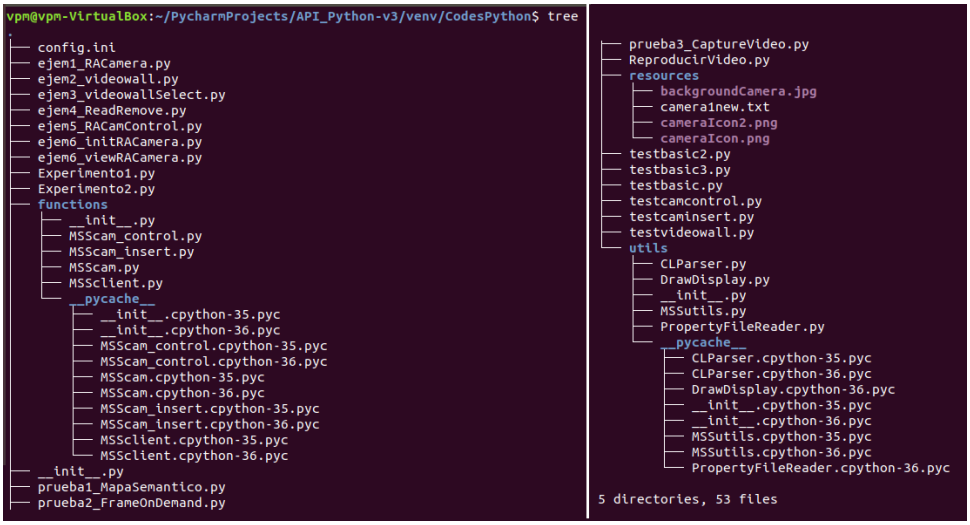


Figura 4.8: Árbol de archivos en CodesPython

De los primeros ejemplos esta el *script* "ejem1_RACamera.py" donde el código ejemplo identifica el número de cámaras existentes en el simulador. Posteriormente se inicializan dichas cámaras, asignándose automáticamente un nombre y un ID las cuales quedan registradas en la API del servidor (ver Figura 4.9). Luego de que cada cámara tiene asignado un ID, el usuario puede seleccionar que cámara desea visualizar en pantalla, para observar su ejecución ir al enlace a continuación: https://vimeo.com/411663813

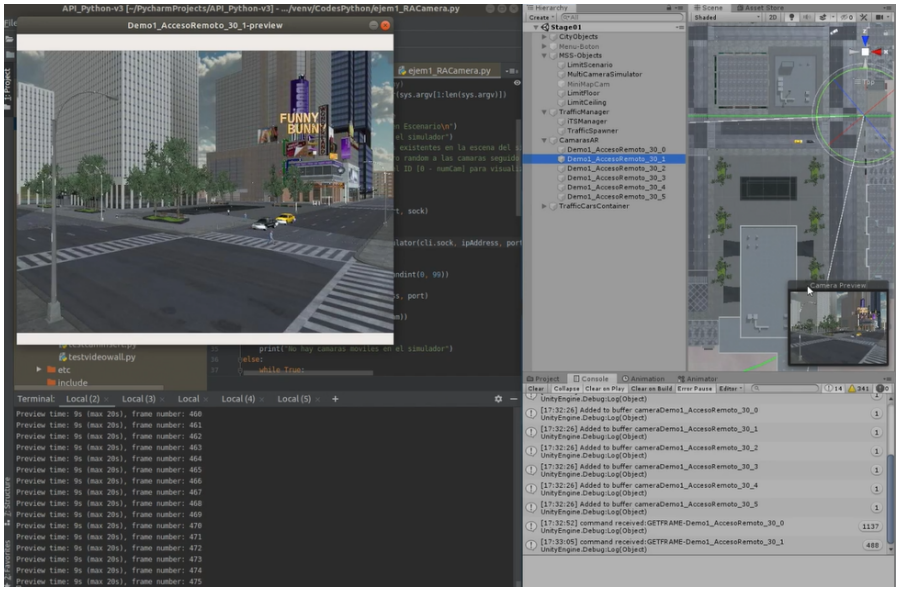


Figura 4.9: Ejecución del *script* "ejem1_RACamera.py" donde se identifica el número de cámaras del escenario y asigna un ID a cada una de ellas.

Cuando un escenario del simulador ya tiene un número de cámaras definidas, mediante la nueva API se puede identificar cuantas existen en la escena, por lo cual en la Figura 4.10 se muestra un panel de varios *display* donde se visualizan las cámaras del escenario. En este caso en concreto el número que se presentan son **6** cámaras. A diferencia del ejemplo anterior, que se visualizan cámaras uno a uno, en este caso se despliegan un panel con las cámaras seleccionadas para visualizarlas. Para mas detalles ver el vídeo a continuación: https://vimeo.com/411664551

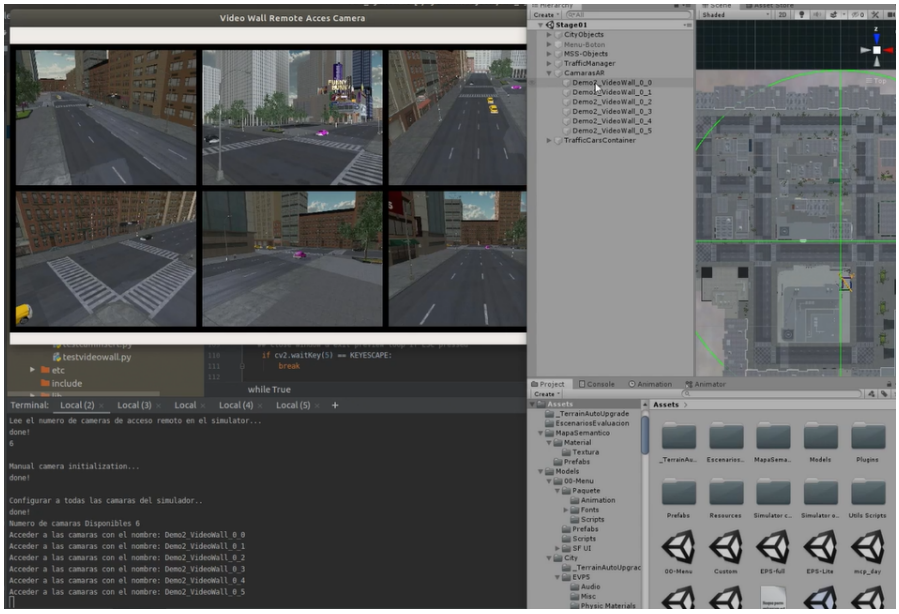


Figura 4.10: Ejecución del *script* "ejem2_videowall.py" para visualizar varias cámaras establecidas en el simulador

Si las cámaras de un escenario es superior a 6, el *script* "ejem3_videowallselect.py" (ver Figura 4.11) permite seleccionar cuales cámaras visualizar en la pantalla, en este caso es de **8** *display*, el usuario a demás puede repetir la visualización de cualquier cámara ya que selecciona las que desea observar. Ver en el enlace el funcionamiento de este ejemplo: https://vimeo.com/411665350

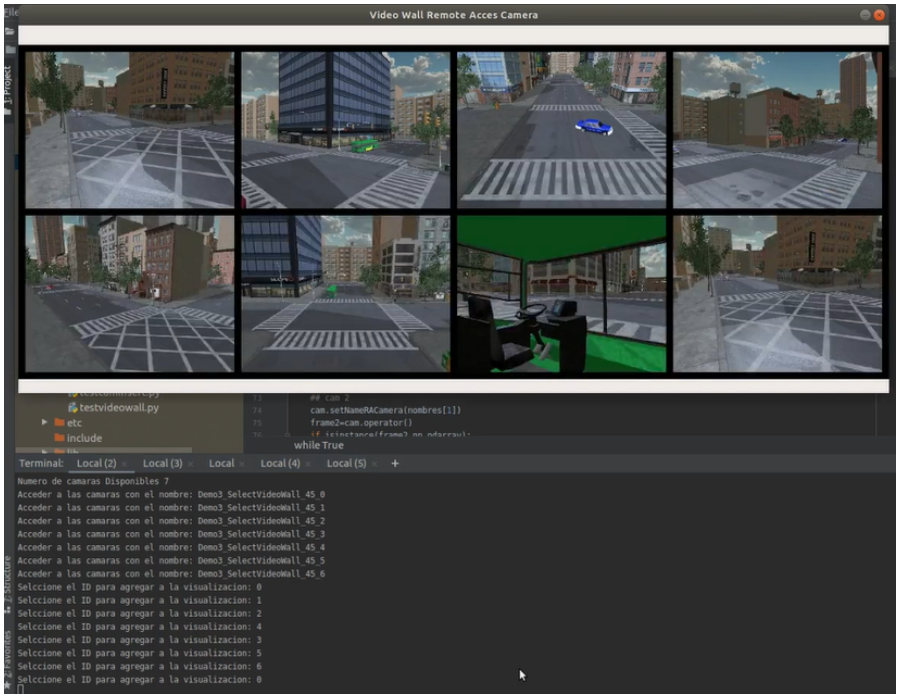


Figura 4.11: En la ejecución del *script* "ejem3_videowallselect.py" el usuario puede seleccionar las cámaras que desea visualizar en el panel de *displays*

Una de las funciones implementadas en la API también permite eliminar los objetos "CameraScene", es decir que una vez identificada las cámaras definidas en el simulador el usuario puede seleccionar la ID de la cámara que desee eliminar definitivamente. Para volver a crear una cámara se puede hacer mediante los *script* "testbasicX.py". Para ver el funcionamiento de este código ir al enlace: https://vimeo.com/411665601

El *script* "ejem5_RACamControl.py" al igual que en el primer ejemplo identifica las cámaras del escenario, los registra con un nombre y asigna un ID, y el usuario selecciona la cámara de desee visualizar. Esta visualización se muestra con una interfaz de usuario para poder **controlar** distintos parámetros de la cámara (movimiento, rotación, traslación). Además los movimientos estarán restringidos según el tipo de cámara que viene establecido desde el simulador (servidor de Unity): *Fixed*, *Mobile*, *PTZ*, mediante este *script* el cliente puede reconocer el tipo de cámara y mostrar el mensaje al usuario. Los detalles el tipo de cámaras se presenta en la sección 3.2. El funcionamiento de este *script* se puede visualizar en: https://vimeo.com/411705357

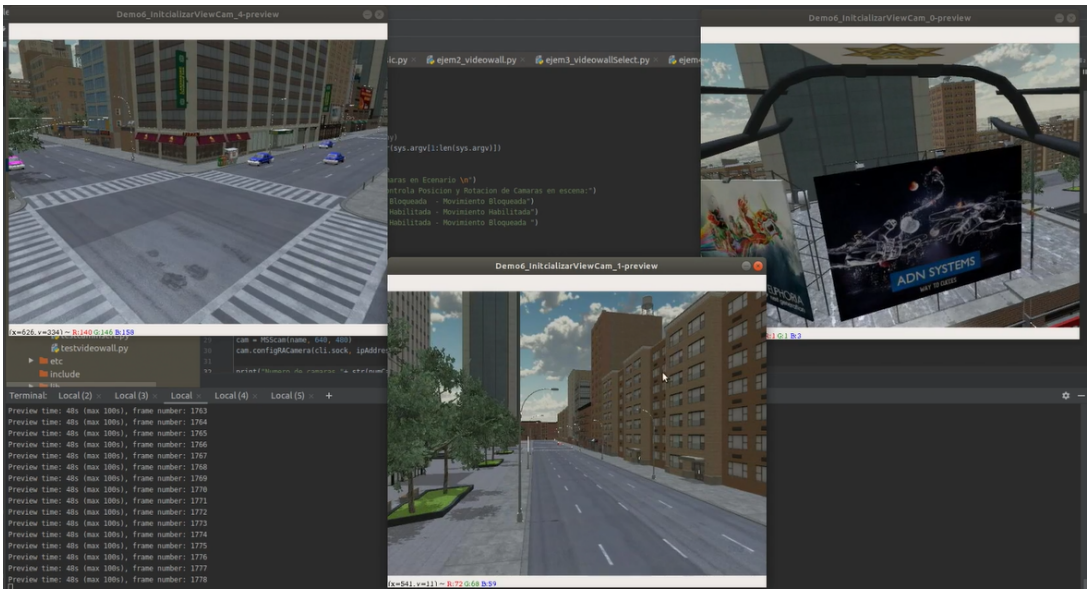


Figura 4.12: Ejecución del *script* "ejem6_initRACamera.py" y "ejem6_viewRACamera.py"

Para el ejemplo 6 se tiene dos *scripts*. El primero "ejem6_initRACamera.py" permite identificar las cámaras del simulador, asignar nombre e ID y registrarlas en el simulador para poder transmitir al cliente, su ejecución se hace una sola vez. El *script* "ejem6_viewRACamera.py" por su parte se puede ejecutar tantas veces como clientes se quieran conectar, ya que hace falta solo hacer la petición de recibir *frames* de las cámaras identificadas del simulador mediante su ID. Para ver el modo de ejecución de este ejemplo ir al vídeo a continuación: https://vimeo.com/411666150

Capítulo 5

Evaluación del rendimiento del simulador

Este capítulo presenta una serie de experimentos desarrollados en base a las distintas variables que se puede encontrar en el simulador por parte del servidor (Unity 3D) como por ejemplo la cantidad de objetos dinámicos y estáticos en un escenario, la cantidad de cámaras que se pueden crear e instalar, el valor de *framerate* de cada cámara para transmitir al cliente, implementado en el capítulo 3, para evaluar el rendimiento del simulador mediante los recursos disponibles del computador in situ.

5.1 Entorno de experimentación

Para la evaluación de los experimentos desarrollados se pone en ejecución el entorno implementado en Unity, el cual se puede variar distintos parámetros de una escena para probar el rendimiento del servidor y/o cliente con todas sus funcionalidades. Dentro de este escenario se puede modificar y adaptar cualquier simulación a necesidad del desarrollador, se define la densidad de objetos (autos y personas) como se requiera para simular tráfico, también es posible colocar el número de cámaras que se desee e independientemente seleccionar que tipo de cámara se inserta (móviles, PTZ, estáticas). Igualmente es posible añadir una capa de personalización a la renderización de las cámaras para obtener un vídeo con su mapa semántico, gran aporte para el estudio de la segmentación semántica dentro de la visión artificial.

El rendimiento del simulador se evalúa en un ordenador personal con las herramientas de hardware especificadas en la Tabla 5.1 a continuación:

Tabla 5.1: Especificaciones de Hardware utilizadas para ejecutar el simulador.

Herramienta	Desarrollador	Propósito
Core(TM) i7-8700 - CPU @3.20GHz - 16GB	Intel(R) UHD	Equipos sobre el que se ejecuta el simulador
GPU GeForce GTX 1070 - 8 GB (8088 MB) procesador x64	NVIDIA	Tarjeta para generar los gráficos (entorno virtual) del simulador

A demás de la utilización de hardware para ejecutar el simulador, en la Tabla 5.2 se muestra los siguientes programas de software y librerías que se emplean para capturar, ordenar y analizar los datos obtenidos.

Tabla 5.2: Especificaciones de Software para captura y análisis de datos del simulador.

Herramienta	Desarrollador	Propósito
Windows 10 Education 64 bits	Microsoft	Sistema operativo para ejecutar el servidor del simulador
Ubuntu 18.04.4 LTS 64 bits	Linux	Sistema operativo para ejecutar el cliente del simulador
Unity v2019.1.7f1	Unity Technologies	Motor de desarrollo de videojuegos para crear escenas.
Python v3.6.9	Python Software Foundation	Lenguaje de programación para el desarrollo de la API del cliente
Archivo Python compilado V3.8.3	Python Software Foundation License	Herramienta para recopilar datos (CPU, GPU, RAM y memoria GPU[VRAM]) en los experimentos.
Matlab	Mathworks	Generar gráficas y calcular estadísticas de los datos recopilados

Para verificar que efectivamente la herramienta con la que se captura y analiza los datos funcione adecuadamente, se utiliza el *software* de *benchmark* "**FurMark**[1]", programa que permite estresar la GPU para verificar su correcto funcionamiento, es así como se verifica que el medidor de rendimiento desarrollada en Python si obtiene los valores verdaderos sobre la usabilidad del la GPU.

5.2 Protocolo de Experimentación

Se define un protocolo para diseñar, evaluar y presentar de forma gráfica y analítica los experimentos que muestran el rendimiento del simulador variando distintos parámetros, esto permite definir un esquema en el cual se pueda basar para desarrollar otros experimentos en futuros cambios y versiones del simulador.

5.2.1 Metodología de evaluación

Para ejecutar, capturar y obtener los datos de los experimentos es importante la definición de la metodología que describa como se va a recopilar la información para posteriormente procesar y obtener resultados. Se definen así algunos factores para la captura de datos así como la estructura del archivo de salida de los mismo para su posterior análisis. Los factores a tomar en cuenta son los tiempos que dura cada experimento, el intervalo de tiempo para obtener cada muestra y el total de muestras obtenidas.

- Duración de cada experimento: **200 s**
- Frecuencia de captura de datos en cada experimento: **0.2 s**
- Muestras obtenidas en cada experimento: $\frac{200s}{0.2s}$=**1000**

Estos parámetros definen la forma en como se van a obtener los datos en cada ejecución, a partir de aquí todos los experimentos diseñados contarán con esta metodología y puede variar para futuras evaluaciones del simulador. Una vez obtenido los datos, éstos se guardan en un archivo plano de texto ".txt" (ver Tabla 5.3) y se presentan de la siguiente forma:

[1] https://geeks3d.com/furmark/

Tabla 5.3: Presentación de datos obtenidos al evaluar cada experimento.

%CPU	**%RAM**	**%GPU**	**%VRAM**	**Etiqueta**
13.1	76.9	5.9	13.7	320x240[DO=10]
5.6	76.9	9.5	13.7	320x240[DO=10]
15.4	76.9	4.7	13.7	320x240[DO=10]
⋮	⋮	⋮	⋮	⋮
7.1	76.9	8.9	13.7	640x480[DO=100]
4.8	76.9	8.6	13.7	640x480[DO=100]
⋮	⋮	⋮	⋮	⋮

5.2.2 Variables de rendimiento

El objetivo de diseñar los experimentos es monitorear y evaluar el rendimiento de los parámetros de usabilidad de CPU y GPU al igual que el consumo en porcentaje de la memoria RAM y VRAM (Video-RAM de GPU).

- CPU.- Del procesador se puede obtener el porcentaje de uso de la CPU, este dato es el valor medio de un procesador **Intel Core i7-8700** con **6 cores** y una frecuencia de **3192.03 MHz**.
- RAM.- La *Random Access Memory* el valor de memoria con la que la CPU funciona, en este caso el valor de RAM es **16 GBytes**.
- GPU.- La tarjeta gráfica **NVIDIA GeForce GTX 1070** con la que se mide el rendimiento del simulador tiene tiene una frecuencia de reloj de **1582 MHz**.
- VRAM.- La **Video-RAM** incorporada con la GPU contiene memoria de tipo **GDDR5 (Samsung)** y un bus de datos de **256 bits**, además el tamaño de memoria es **8192 MB (8GB)** y *Bandwidth* de **256.3 GB/s**.

5.2.3 Identificación de variables del simulador

En la ejecución de los experimentos dentro del escenario a evaluar se ha previsto un gran conjuntos de variables para estructurar y estimar el rendimiento del simulador, estas variables dinámicas permiten combinar una gran cantidad de distintos experimentos para formular tantos casos de evaluación como se requiera.

Los parámetros que se pueden evaluar a lo largo de los experimentos son:

- Calidad (Q).- Donde Unity establece el nivel de calidad predeterminado y adecuado utilizado para cada plataforma donde se ejecute el escenario, además las opciones dependen de los recursos del ordenador y el monitor donde se ejecute el simulador.
- Conexión de Clientes.- El cual va ha estar delimitado a una sola conexión de cliente, es decir dentro del mismo ordenador se ejecutara el servidor y la conexión de un solo cliente.
- Densidad de Objetos (DO).- El valor que determina la generación aleatoria de objetos animados (personas y automóviles) dentro de una escena.
- Cámaras de Escena (CS).- Número de cámaras pre-definidas dentro de cada escenario de experimentación, estos pueden ser cámaras RGB y/o con mapa semántico.

- Cámaras de Usuario (CU).- Es la cantidad de cámaras que se pueden crear y conectar como clientes en el simulador.
- *Framerate* (FPS).- Valor con el que se puede oscilar para la transmisión de fotogramas por segundo seleccionado por el cliente.
- Resolución (Res).- Resolución establecida a las cámaras para transmitir del servidor al cliente.

Los parámetros que se observan en la Tabla 5.4 se establecen en un cierto valor y cantidad para considerar al momento de evaluar, estos se pueden combinando para realizar los diferentes configuraciones de experimentación.

Tabla 5.4: Identificación de variables para evaluación

	Acrónimo	Bajo	Medio	Alto
Calidad	Q	*Fastest*	*Simple*	*Fantastic*
Densidad de Objetos	DO	10	100	1000
Cámaras de Escena[2]	CS	2	10	30
Cámaras de Usuario[3]	CU	1	3	6
FrameRate cámaras	FPS	10	20	30
Resolución	Res	320×240 (QVGA)	640×480 (VGA)	1280x720 (HD)

5.2.4 Herramienta de Medición

La herramienta que se utiliza para supervisar el uso de los recursos y los procesos del ordenador, es un archivo compilado de Python (.pyc) (por la facilidad de tener un archivo ejecutable), es importante resaltar que este archivo se lo puede ejecutar dentro de un entorno Windows que tenga instalado el paquete de Python en su versión 3.8.3 ya que ésta es la versión de compilación, es importante aclarar que esta versión es independiente de la API de Python del cliente con la cual se ha trabajado a lo largo del TFM.

La recopilación de datos permite obtener algunos de los parámetros más importantes para analizar el rendimiento del simulador, estos son el %Uso de CPU, % de memoria RAM, % de Uso de la GPU y % de memoria de GPU (VRAM). La pantalla de ejecución de esta herramienta de medición se puede observar en la Figura 5.1 además a continuación se puede acceder al siguiente enlace para comprender su fácil funcionamiento: https://vimeo.com/423368542

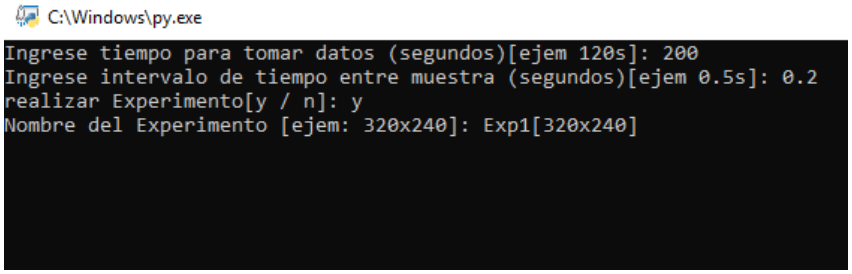


Figura 5.1: Datos solicitados para ejecutar la herramienta de medición

[2]Cámaras pre-establecidas en la escena, dentro del cual existirán cámaras fijas o sobre objetos móviles
[3]Cámaras que el usuario crea a partir de la API

5.3 Desarrollo de experimentos sobre la interfaz de Unity

En este apartado se desarrolla distintas configuraciones según las variables descritas en la Tabla 5.4 para ejecutar varios escenarios del simulador.

Las distintas condiciones a evaluar a lo largo de esta sección se compara el rendimiento de cada experimento del simulador frente a los recursos del equipo cuando se encuentra en reposo o *standby*. Para poner a prueba los dichos recursos:

- En el **primer experimento** se configura de forma tal que se define una **resolución (RES) baja** y se varía el parámetro de **densidad de objetos (DO)**.
- Para el **segundo experimento** se sube la **resolución (Res) a media**, se selecciona la **calidad (Q) en "Simple"** y se varía **densidad de objetos (DO)**.
- El **tercer experimento** se aumenta la **resolución (Res) a la más alta**, se sube la **calidad (Q) en "Fantastic"** y se mantiene variando **densidad de objetos (DO)** para cada experimento.
- En el **cuarto experimento** se ha seleccionado mantener la **resolución (Res) Alta**, fijar la **densidad de objetos (DO) en media** y variar los **FPS**.
- En el **quinto experimento** la configuración que se diseña es variar el número de **cámaras de escena (CS)** dentro del por cada cámara RGB existe otra homóloga con filtro de mapa semántico.

5.3.1 Resolución baja con variaciones de objetos en escena

Este experimento analiza en primera instancia el rendimiento del ordenador personal bajo condiciones donde no se ejecuta ni se hace uso del simulador ni otro programa para evaluar la usabilidad de CPU y GPU en estado de reposo, posteriormente se realiza los experimentos partiendo de la construcción de los escenarios probado en el editor de Unity donde se cambia la cantidad de objetos existentes en el entorno "ciudad", es decir generar una cantidad determinada de vehículos y personas según la variable DO que se puede observar en la Tabla 5.5. Además se presenta en la Figura 5.2 imágenes del experimento al aumentar la densidad de objetos en escena.

Tabla 5.5: Distintas configuración de densidad de objetos

Resolución	Densidad de Objetos	FrameRate	Cámaras Usuario	Calidad
Res = Baja	**DO = Variable**	FPS = Baja	CU = Baja	Q = Baja
320x240	**10**	10	1	*Fastest*
320x240	**100**	10	1	*Fastest*
320x240	**1000**	10	1	*Fastest*

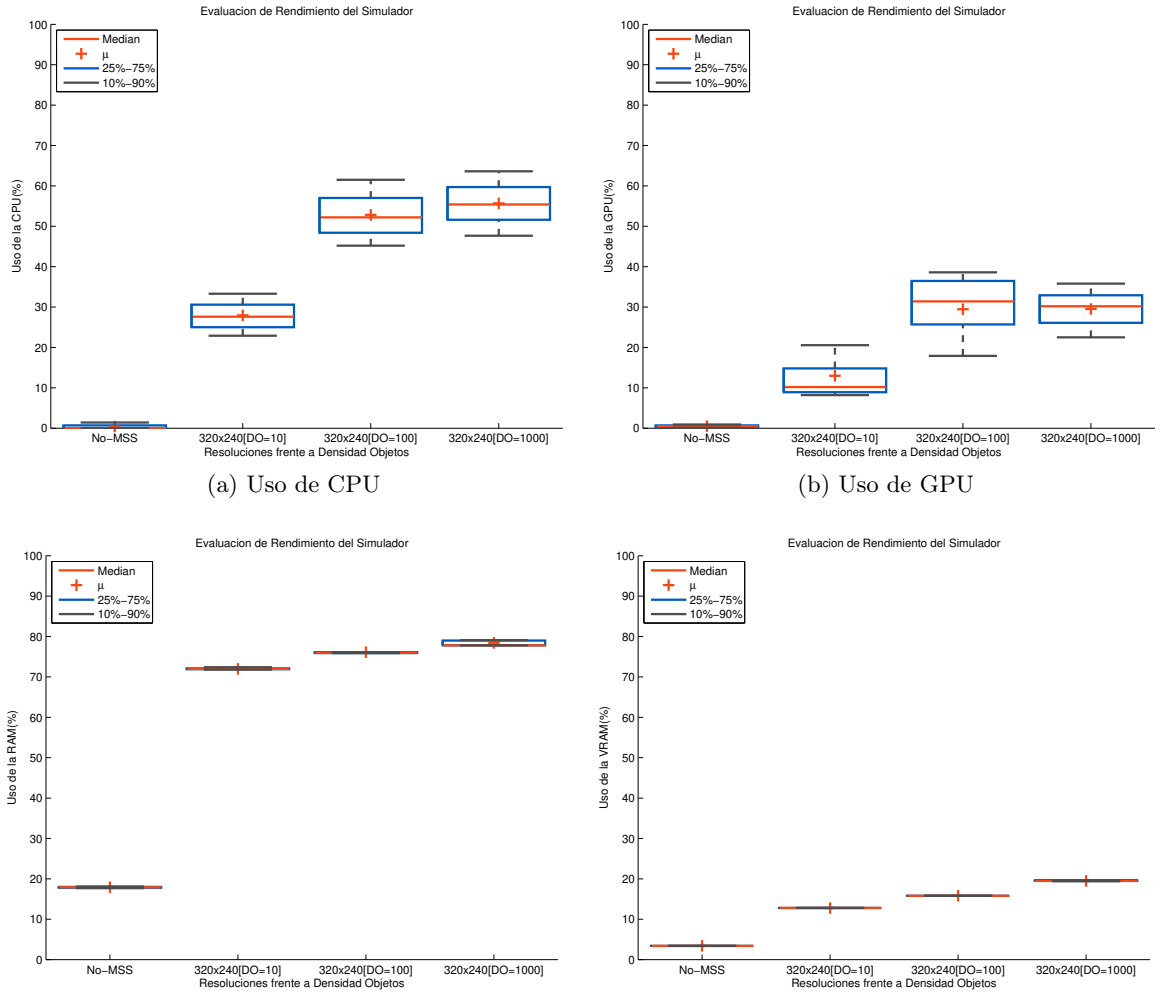
(a) Densidad baja (b) Densidad media (c) Densidad alta

Figura 5.2: Variación de densidad de objetos en escena del simulador

En la Figura 5.3a se puede observar que el uso de la CPU al recolectar datos cuando no se ejecuta el simulador es significativamente inferior a los experimentos con las diferentes configuraciones. Se puede apreciar que la usabilidad de CPU aumenta cuando el simulador requiere procesar el aumento de objetos dentro del escenario. A diferencia de lo que se puede analizar sobre el rendimiento de la GPU, observando la Figura 5.3b la variación es mínima en los tres experimentos, exceptuando el modo "No-MSS" ya que en estos casos la resolución de todos se mantiene siempre la misma (320x240 y *Fastest*)

En el Figura 5.3c se observa el el porcentaje que hace uso la memoria RAM ligada a la CPU donde llega alrededor del 80% para un DO alto, mientras que a diferencia de éste, la VRAM (ligada a la GPU, Figura 5.3d) rodeando el 20% de uso aumenta en cada experimento debido a que esta muy enfocado las gráfica y renderizado de texturas.

El argumento de las gráficas denominado **No-MSS** es el porcentaje de uso de las variables de rendimiento cuando el ordenador esta en reposo, es decir que se hace una medida del rendimiento cuando no se ejecuta el simulador o algún otro programa que interfiera con las valores capturados.



(a) Uso de CPU

(b) Uso de GPU

(c) Uso de RAM

(d) Uso de VRAM

Figura 5.3: Uso de recursos establecido en mínima resolución variando la densidad de objetos

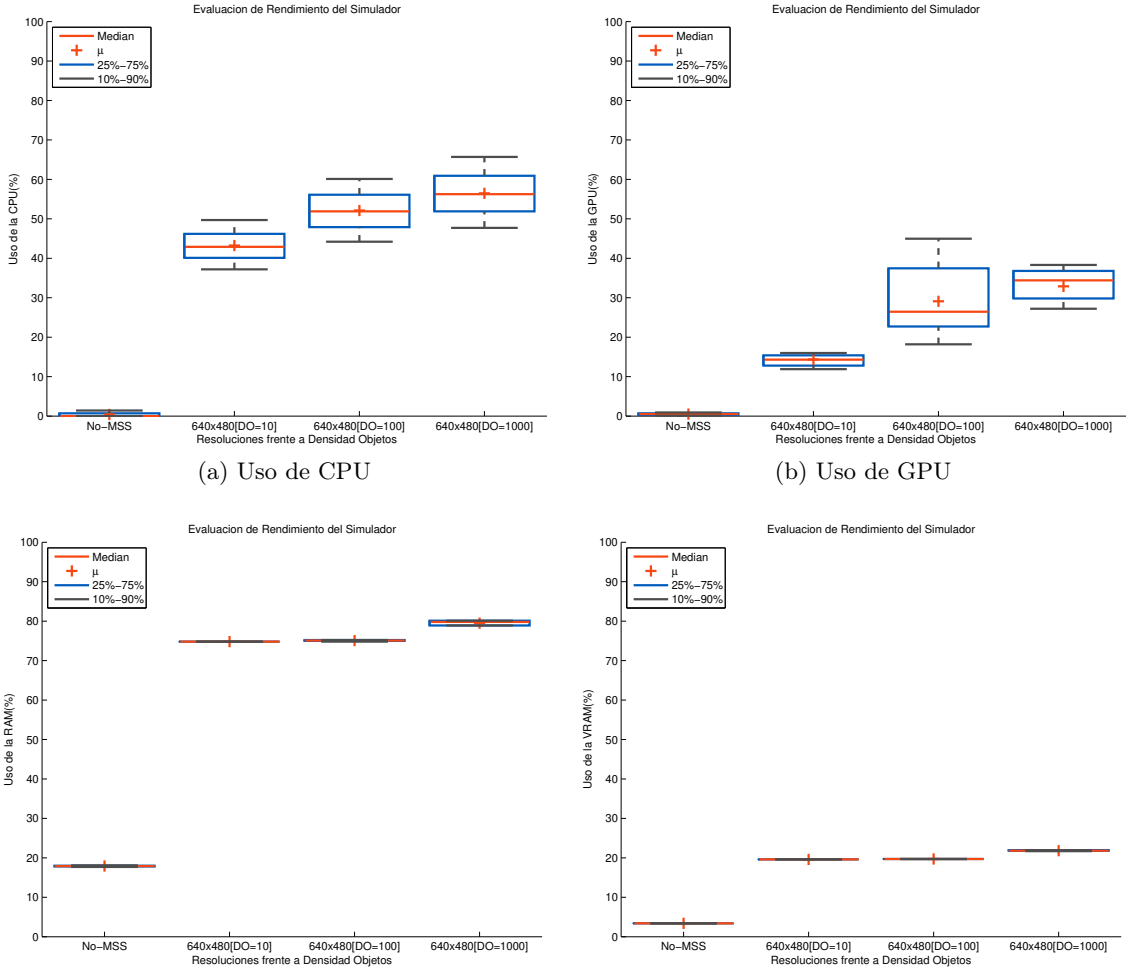
5.3.2 Resolución media con variaciones de objetos en escena

Tabla 5.6: Resolución y calidad media variando densidad de objetos

Resolución	Densidad de Objetos	FrameRate	Cámaras Usuario	Calidad
Res = Media	**DO = Variable**	FPS = Baja	CU = Baja	Q = Media
640x480	**10**	10	1	*Simple*
640x480	**100**	10	1	*Simple*
640x480	**1000**	10	1	*Simple*

Para este experimento se parte modificando la resolución en la que se ejecuta el simulador, el caso concreto es una resolución media de 640x480 (VGA), adicionalmente la calidad para presentar los gráficos es media

("*Simple*") y se mantiene cambiando la variable densidad de objetos. Los parámetros exactos de dicho experimento se puede observar en la Tabla 5.6.



(a) Uso de CPU

(b) Uso de GPU

(c) Uso de RAM

(d) Uso de VRAM

Figura 5.4: Rendimiento con una resolución media y variando la densidad de objetos

En la Figura 5.4a se denota que a una resolución mayor y aumentando la cantidad de objetos que el simulador crea en una escena la usabilidad de CPU crece llegando a tener una usabilidad de casi el 60%. Por otro lado se observa que en esta resolución designada (media) la GPU se comporta distinto en cuanto a su usabilidad, ya que para este caso el porcentaje mostrado en la Figura 5.4b no depende de la variable "densidad de objetos", al solicitar al simulador una sola cámara de usuario significa que solo se transmite los *frames* en una pantalla y al designar la resolución 640x480 se presenta dicho comportamiento.

En la Figura 5.4c tiene un comportamiento similar cuando se ejecuta los tres experimentos a diferencia del "No-MSS" esto debido a que la memoria RAM es solo la parte accesible de la CPU por lo que en todos los procesos importantes para ejecutar esta configuración lo sufre la CPU. A diferencia de esto la VRAM es quien se encarga de manejar los gráficos de la GPU por lo cual esta variación aumenta levemente al subir la resolución de las imágenes (observar en la Figura 5.4d).

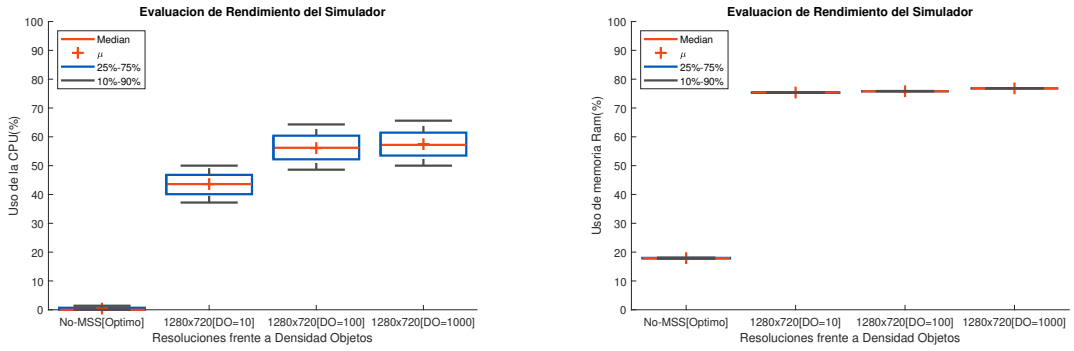
5.3.3 Resolución alta con variaciones de objetos en escena

En esta configuración se aumenta a una resolución más alta y colocando a la mejor calidad de gráficos, como en los casos anteriores el parámetro que se modifica es la de densidad de objetos así se pretende definir que resolución es la más adecuada para cambiar otras variables posteriormente, para este experimento se puede observar la configuración en la Tabla 5.7.

Tabla 5.7: Resolución y calidad alta variando densidad de objetos

Resolución	Densidad de Objetos	FrameRate	Cámaras Usuario	Calidad
Res = Alta	**DO = Variable**	FPS = Baja	CU = Baja	Q = Alta
1280x720	**10**	10	1	*Fantastic*
1280x720	**100**	10	1	*Fantastic*
1280x720	**1000**	10	1	*Fantastic*

Se observa nuevamente que los datos recolectados cuando no se usa el simulador ("No-MSS") que la CPU y GPU tiene una usabilidad significativamente baja a diferencia de cuando se evalúa cualquier entorno con el simulador. En la Figura 5.5a se observa que en el experimento de 1280x720 [DO=10] es ligeramente un 10% menos que en los otros dos debido en estos últimos la CPU debe generar gran cantidad de objetos.



(a) Uso de CPU

(b) Uso de RAM

Figura 5.5: Rendimiento CPU y RAM con una resolución alta y variando la densidad de objetos

En el caso de la RAM como se observa en la Figura 5.5b varia el experimento 1280x720 [DO=100] pudiéndose deber a que la visualización de la cámara abarca un mayor campo visual con una mayor resolución por lo que al presentar los gráficos en el cliente se observa mayor afluencia de objetos, eso hace que la memoria RAM trabaje de forma similar para objetos de 100 y 1000 ya que en una sola pantalla esta diferencia de objetos no es apreciable, es decir que para que exista un cambio, los 1000 objetos deberían estar en el mismo punto donde enfoca la cámara y no distribuidos por toda la escena tal como lo hace el simulador.

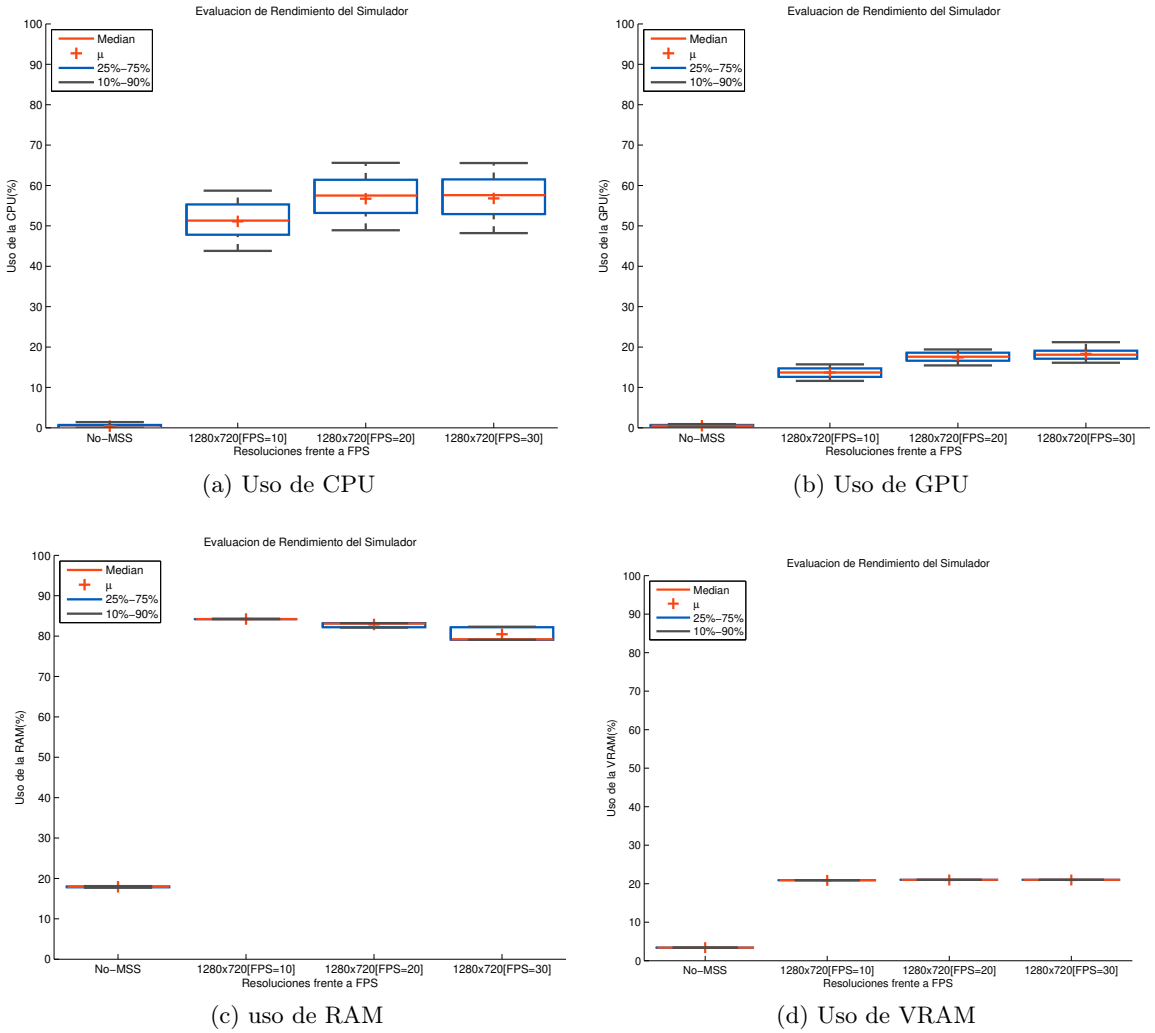
5.3.4 Distinta transmisión de FPS

Con un análisis basado en los tres experimentos anteriores se determina establecer en fijo las siguientes variables: densidad de objetos en media (DO=100), Calidad alta (Q=*Fantastic*) y resolución de pantalla

en alta (Res=1280x720) para de esta forma evaluar el rendimiento al ir variando el parámetro de *framerate* para lo cual la configuración empleada en este experimento se puede observar a detalle en la Tabla 5.8.

Tabla 5.8: Variación del parámetro de resolución

Resolución	**Densidad de Objetos**	**FrameRate**	**Cámaras Usuario**	**Calidad**
Res = Alta	DO = Media	**FPS = Variable**	CU = Baja	Q = Alta
1280x720	100	**10**	1	*Fantastic*
1280x720	100	**20**	1	*Fantastic*
1280x720	100	**30**	1	*Fantastic*



(a) Uso de CPU

(b) Uso de GPU

(c) uso de RAM

(d) Uso de VRAM

Figura 5.6: Rendimiento de CPU y GPU fijando una resolución alta y variando los FPS

En la Figura 5.6a se puede observar que el uso de la CPU en esa resolución consume gran cantidad

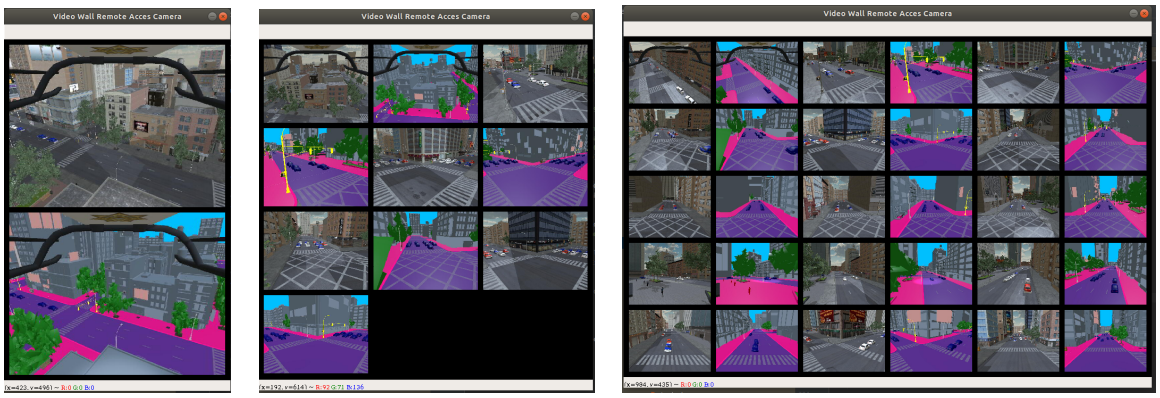
de recursos ya que con una mayor resolución y calidad de gráficos el factor FPS es muy importante para conseguir un buen rendimiento del simulador. Por lo que es importante contar con una buena tarjeta gráfica ya que la GPU es la encargada de estos procesos de vídeo.

A la par con la CPU la memoria RAM es un factor muy importante para el funcionamiento de los procesos como en la Figura 5.6c la tendencia en la gráfica llega casi a saturar la memoria RAM y con lo observado en los experimentos el caso de 1280x720 [FPS=20] y 1280x720 [FPS=30] no llega a transmitir los FPS efectivos solicitados, el valor de FPS que se envía desde el simulador **oscila entre 19 y 20 FPS**. En el ultimo caso no se envían los 30 FPS efectivos, al realizar el experimento se comprueba que la transmisión queda en un valor de entre 15 y 16 FPS por lo cual representa un 50% de la tasa de *frame* menos de lo que el cliente configura.

En la Figura 5.6b se observa un cambio diferencial en el caso de 1280x720 [FPS=10] con respecto a cuando se aumenta los FPS, debido a que el simulador funciona mejor al transmitir entre 10 y 20 FPS ya que aunque la GPU esta preparada para gráficos de alta calidad, la licencia de Unity que que se usa para estas simulaciones al ser gratuita tiene muchas limitaciones, y en este caso al seleccionar FPS mayor a 20 realmente no es lo que entrega el servidor. Para la memoria VRAM como se observa en la Figura 5.6c la variación es mínima ya que aun con los juegos más actuales la VRAM no ocupa en su totalidad brindando así una gran fidelidad para correr gráficos.

5.4 Desarrollo de experimentos sobre el simulador compilado

La disposición para visualizar el número de cámaras se establece en una panel con varias pantallas en una resolución de 640x480, el parámetro que va a ir variando para este experimento son las cámaras de escena y se puede observar en la Figura 5.7 cada caso concreto para obtener los datos de evaluación.



(a) CS bajo (b) CS medio (c) CS alto

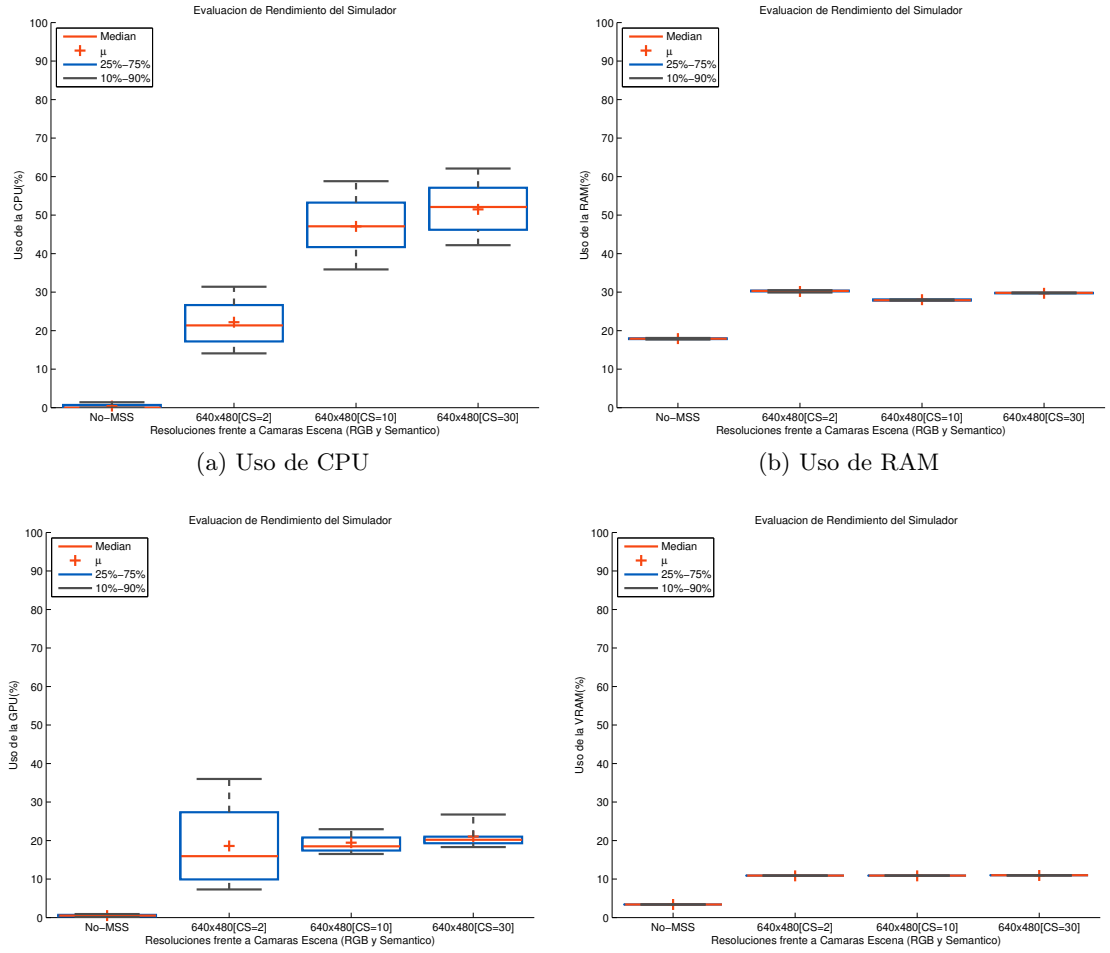
Figura 5.7: *Display* de cámaras de usuario en RGB y su homólogo con mapa semántico

Después de diseñar varios experimentos sobre la interfaz de Unity, se construye 3 archivos compilados del simulador donde la resolución, la densidad de objetos, los FPS y la calidad se mantiene fijos tal como se puede observar en la Tabla 5.9.

Tabla 5.9: Variación del parámetro de resolución

Resolución	Densidad de Objetos	FrameRate	Cámaras de escena		Calidad
Res = Media	DO = Media	FPS = Baja	**cámara RGB**	**mapa semántica**	Q = Media
640x480	100	10	**1**	**1**	*Simple*
640x480	100	10	**5**	**5**	*Simple*
640x480	100	10	**15**	**15**	*Simple*

5.4.1 Compilación de varios escenarios aumentando *CameraScene*



(a) Uso de CPU

(b) Uso de RAM

(c) Uso de GPU

(d) Uso de VRAM

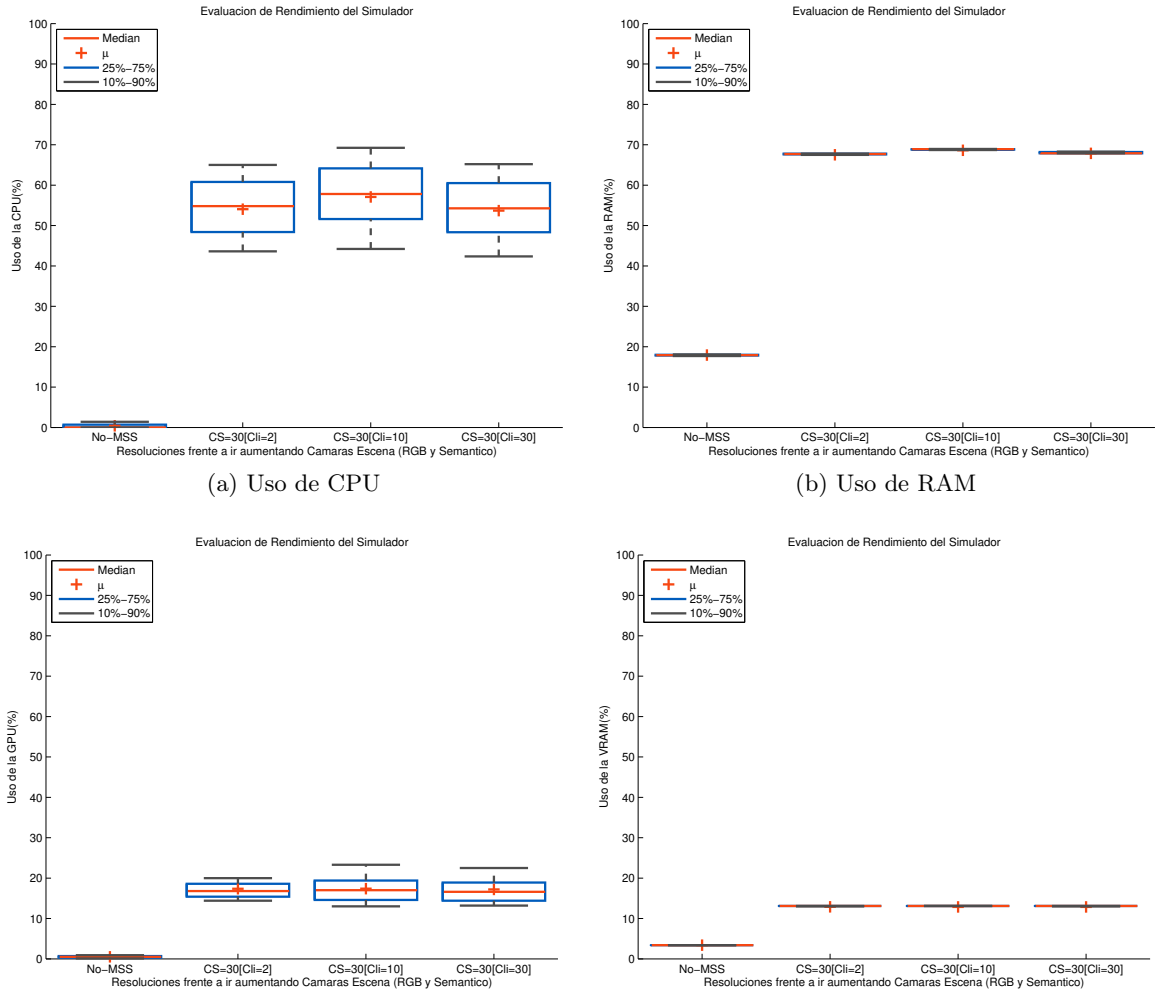
Figura 5.8: Rendimiento fijando una resolución media y simulador compilado con 2, 10 y 30 cámaras de escenario (*CameraScene*)

En esta evaluación se compila **3** escenarios distintos, cada uno de ellos va aumentando el número de *CameraScene* (pasando de 2 a 10 hasta 30 cámaras), para lo cual por cada cámara RGB que se coloque en el simulador a la par tendrá otra cámara de similares características pero con su capa de mapa

semántico, por lo que la variable cámaras de escena (CS) llegará a un máximo de 30 (15 RGB y 15 con mapa semántico) las cuales estarán distribuidas por la escena.

En la evaluación del rendimiento de recursos para este experimento se puede observar en las Figuras 5.8a, 5.8c el caso 640x480[CS=2] es un escenario compilado con dos cámaras de tipo *CameraScene*, donde su rendimiento de CPU y GPU es mejor que en los dos casos siguientes presentando un porcentaje de uso de la CPU de 20% y GPU del %15 (aproximadamente), es importante destacar que para este caso la tasa de *frames* que entrega el simulador es **FPS=30**. En los casos siguientes, 640x480[CS=10] y 640x480[CS=30] claramente se observa que el porcentaje de uso aumenta ya que la variable **CS *CameraScene*** también aumenta, sin embargo la cantidad de FPS que se observa en el simulador se reduce a **FPS=27 y FPS=11** respectivamente. Las Figuras 5.8b, 5.8d tiene similar rendimiento en todos los casos, esto debido a que el simulador solo ejecuta una ventana lo que conlleva a que las resoluciones de las imágenes sean las mismas.

5.4.2 Compilación de un escenarios con valor máximo de *CameraScene*



(a) Uso de CPU

(b) Uso de RAM

(c) Uso de GPU

(d) Uso de VRAM

Figura 5.9: Rendimiento del simulador compilado con 30 cámaras de escena (*CameraScene*) llamando a 2, 10 y 30 instancias de cámaras desde cliente

Para este experimento, se ha utilizado un solo compilado de la escena. Este archivo ejecutable es el que se construyo con 30 cámaras de escenario (CS=30, RGB=15 y Semántica=15), en este experimento el cliente instancia desde un *script* la visualización de: 2, 10 y posteriormente 30 cámaras. Por lo cual a cada caso se lo ha denominado "CS=30[Cli=2,10,30]" (simulador compilado = 30 cámaras [visualización del cliente = X cámaras]).

Como se puede observar en las Figuras 5.9a y 5.9c El rendimiento del simulador es similar ya que en los tres casos el archivo compilado es el mismo, este rendimiento se mantiene dentro de los mismos parámetros como el experimento anterior con el caso 640x480[CS=30] ya que es el mismo ejecutable. Por otro lado en las Figuras 5.9b y 5.9d el rendimiento aumenta ya que en este caso al usar al cliente para visualizar las cámaras se necesita un mayor porcentaje de RAM y VRAM, lo que se ve reflejado en el porcentaje de uso. Es importante destacar que para estos tres casos se declaro la variable **FPS=10** pero observando en el experimento esto no es así, la cantidad de FPS que refleja el simulador es de entre **5-6 FPS** por lo que se observa una clara reducción de la tasa de *frames*.

Al diseñar los experimentos y obtener los resultados se puede observar que los valores de usabilidad de la tarjeta gráfica son bajos, es decir que se esperaría estresar la GPU al aumentar la complejidad de la escena, sin embargo esto no sucede a pesar de haber desarrollado escenas con mas carga de objetos o resoluciones altas. Se puede interpretar según los resultados que; la GPU llega a tener un rendimiento de usabilidad de entre el **10% al 30%** mientras que el cuello de botella se presenta sobre la CPU. El experimento de la Figura 5.8 donde no se instancia cámaras ni captura de datos presenta un rendimiento simular en la GPU que en el experimento de la Figura 5.9 donde sí se esperaría estresar la GPU ya que se instancian todas las cámaras y el *buffer* de cada cámara aumenta, hecho que no sucede.

Capítulo 6

Aplicación: Segmentación Semántica

Este capítulo introduce el algoritmo seleccionado para el uso de la segmentación semántica uno de los tantos existentes en la visión artificial para aplicarlo sobre datos obtenidos del simulador de las escenas desarrolladas. Finalmente se presentan distintos resultados de imágenes procesadas por el dicho algoritmo.

6.1 Algoritmo seleccionado

En la parte de aplicación del trabajo de fin de máster con las mejoras del simulador se ha desarrollado el uso de la segmentación semántica con OpenCV y el aprendizaje profundo por lo cual, dentro de esto, es importante definir el modelo en la que se base el algoritmo ejemplo, en este caso en concreto es la arquitectura ENet.

ENet (red neuronal eficiente) es una propuesta de arquitectura de red neuronal profunda, desarrollada específicamente para tareas que requieren baja latencia. **ENet** permite realizar una segmentación semántica de píxeles en tiempo real, es 18 veces mas rápida, requiere 75 menos FLOP, tiene 79 parámetros menos y proporcionando una precisión similar o mejor a modelos existentes [17].

Mediante esta arquitectura es posible aplicar la segmentación semántica a imágenes y transmisión de vídeo. La arquitectura de la red se observa en la Figura 6.1 (Tabla de la izquierda) se encuentra dividida en etapas y se describe los tamaños de salida resultantes para una resolución de imagen de entrada de ejemplo de 512 x 512. Además se observa en la Figura 6.1 (derecha) los bloques que tiene la etapa inicial. El algoritmo seleccionado para ésta aplicación adopta una vista de ResNets que describe una sola rama principal y extensiones con filtros convolucionales. Posteriormente se fusionan en un elemento, cada bloque consta de tres capas convolucionales: una proyección 1x1 que reduce la dimensionalidad, una capa convolucional principal y una expansión 1x1 [17].

Name	Type	Output size
initial		16 × 256 × 256
bottleneck1.0	downsampling	64 × 128 × 128
4× bottleneck1.x		64 × 128 × 128
bottleneck2.0	downsampling	128 × 64 × 64
bottleneck2.1		128 × 64 × 64
bottleneck2.2	dilated 2	128 × 64 × 64
bottleneck2.3	asymmetric 5	128 × 64 × 64
bottleneck2.4	dilated 4	128 × 64 × 64
bottleneck2.5		128 × 64 × 64
bottleneck2.6	dilated 8	128 × 64 × 64
bottleneck2.7	asymmetric 5	128 × 64 × 64
bottleneck2.8	dilated 16	128 × 64 × 64
Repeat section 2, without bottleneck2.0		
bottleneck4.0	upsampling	64 × 128 × 128
bottleneck4.1		64 × 128 × 128
bottleneck4.2		64 × 128 × 128
bottleneck5.0	upsampling	16 × 256 × 256
bottleneck5.1		16 × 256 × 256
fullconv		C × 512 × 512

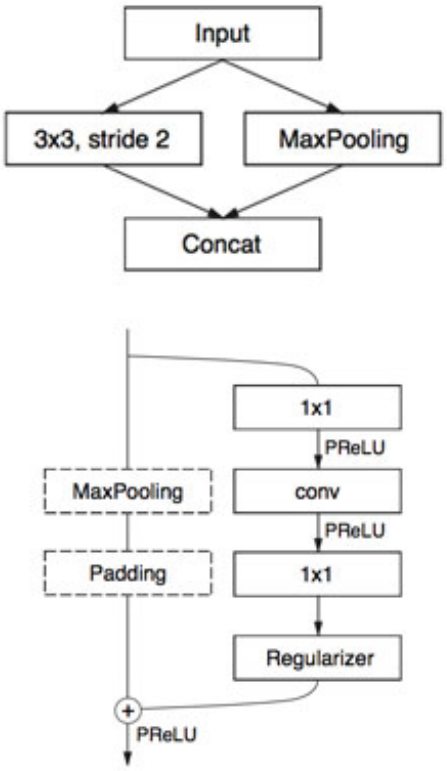


Figura 6.1: Arquitectura ENet. Tamaños de salida dados para una entrada ejemplo de 512x512 y estructura de bloques convolucionales de las etapas [17].

6.1.1 Descripción

El modelo en particular que se ha usado esta entrenado en 20 clases que incluyen:

- Sin etiquetar (fondo)
- Carretera
- Acera
- Edificio
- Pared
- Cerca
- Poste
- Semáforo
- Señal de tráfico
- Vegetación
- Terreno
- Cielo
- Persona
- Ciclista
- Automóviles
- Camión
- Autobús
- Tren
- Motocicleta
- Bicicleta

El directorio actual con el que se realiza el ejemplo de esta aplicación contiene la siguiente estructura de árbol (Figura 6.2) y se divide en 4 directorios.

- `enet-cityscapes`: Contiene el modelo de aprendizaje profundo pre-entrenado, lista de clases y etiquetas de color correspondiente a las 20 clases (número correspondiente a la red escogida que se ajusta con las 12 clases existentes en el simulador), para que este ejemplo de segmentación semántica se ajuste al protocolo del capitulo 3.3 se ha duplicado el archivo **enet-colors.txt** por **unity-colors.txt**[1] el cual tiene las etiquetas de colores mostrados en la Tabla 3.2.

[1]Se puede renombrar el archivo .txt por **mss-color.txt** y tenerlo en cuanta al ejecutar los *scripts* de la red

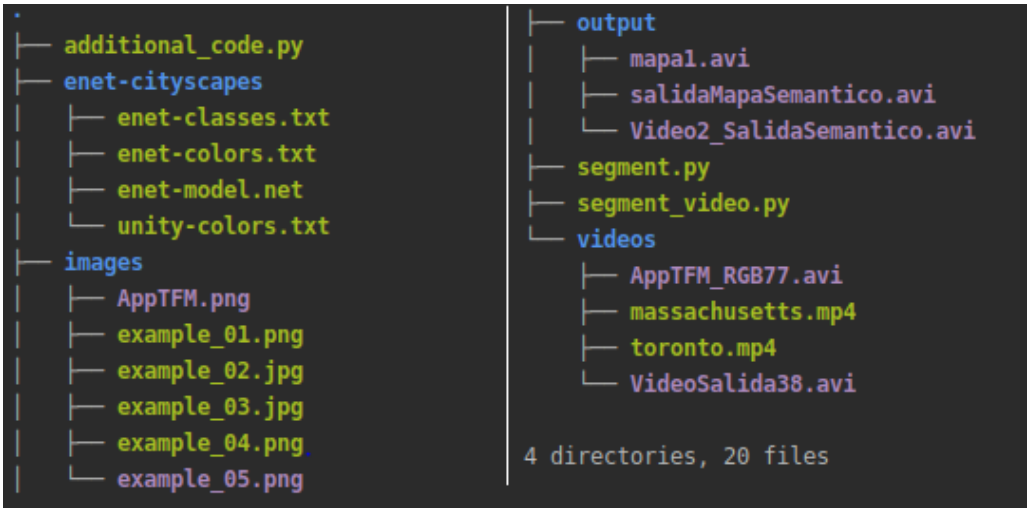


Figura 6.2: Estructura de árbol del proyecto.

- `images`: Son archivos de imágenes de muestra o ejemplos sobre los cuales se prueba el algoritmo, concretamente el *script* **"segment.py"**.
- `Output`: Para fines de organización se ha creado el fichero "salida" para que al ejecutar los ejemplos del algoritmo, el video procesado se guarde en dicha carpeta.
- `videos`: En esta carpeta se deben guardar los vídeos de muestra para probar el *script* **"segment_video.py"**, hay dos ficheros que vienen con el proyecto y son de vídeos reales, y otros dos archivos (.avi) que se han obtenido del simulador.

6.1.2 Configuración

Es prescindible importar los paquetes necesarios para la ejecución de los *script*, en la Figura A.2 se muestra los requerimientos que incluyen los paquetes para el funcionamiento de la segmentación semántica de este proyecto. Si es necesario también se recomienda el uso de entornos virtuales de Python descrito en la sección A.1.

El *script* **segment.py** es el ejemplo para aplicar la segmentación semántica sobre imágenes de ejemplo y contiene los siguientes 5 argumentos, de los cuales 2 son opcionales:

- `--model`: seleccionar la ruta donde se encuentra el archivo **enet-modelo.net** que es el modelo que se ha elegido.
- `--classes`: el parámetro es un archivo de texto, y se selecciona la ruta al archivo que contiene las etiquetas de las clases (ejem: *Unlabeled, Road, Sidewalk* ...)
- `--image`: Ruta de archivo de la imagen de entrada (la que se va a testear).
- `--colors`: OPCIONAL, ruta a un archivo de texto donde se especifica los colores de las clases que se listaron en **enet-classes.txt**, si no se especifica ningún archivo, se asignará colores aleatorios a cada clase.
- `--width`: OPCIONAL, indica el ancho de la imagen deseado, por defecto el valor es 500 píxeles.

```
1 ap = argparse.ArgumentParser()
2 ap.add_argument("-m", "--model", required=True,
3   help="path to deep learning segmentation model")
4 ap.add_argument("-c", "--classes", required=True,
```

```
    help="path to .txt file containing class labels")
ap.add_argument("-i", "--image", required=True,
    help="path to input image")
ap.add_argument("-l", "--colors", type=str,
    help="path to .txt file containing colors for labels")
ap.add_argument("-w", "--width", type=int, default=500,
    help="desired width (in pixels) of input image")
args = vars(ap.parse_args())
```

La aplicación muestre 3 ventanas resultantes, como se observa en la Figura 6.3. Una imagen donde se muestra las etiquetas correspondientes antes descritas, la imagen original de entrada y la salida del algoritmo, para observar mayor detalle en la ejecución ver el siguiente enlace: https://vimeo.com/407031600.

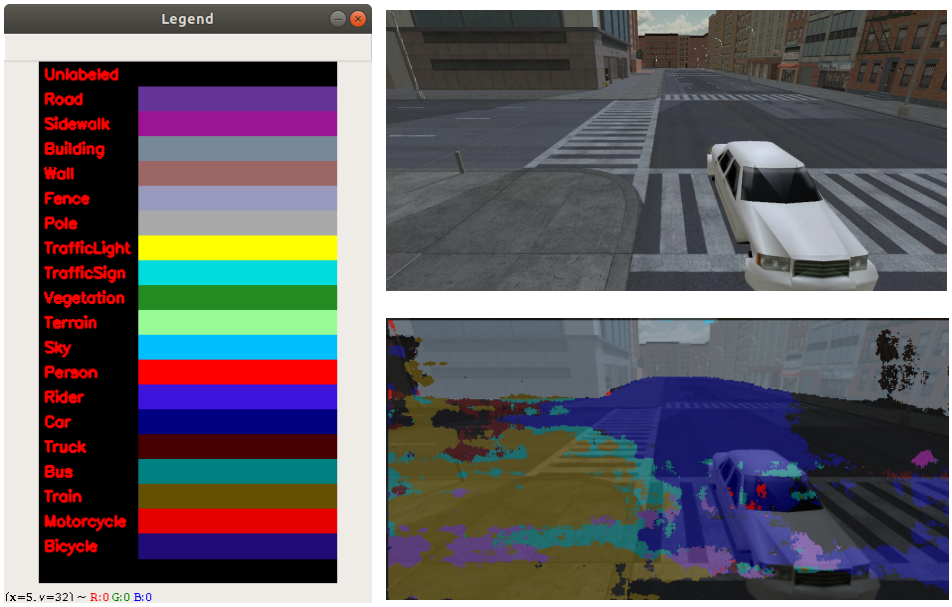


Figura 6.3: Escenario de prueba y etiqueta de clases

Dentro de las soluciones dotadas al simulador es la de renderizar un mapa semántico a través de la API de Python o en la configuración básica de las cámaras del simulador por lo cual es posible tener el *ground-truth* con el cual comparar los resultados de cualquier algoritmo (ver Figura 6.4).

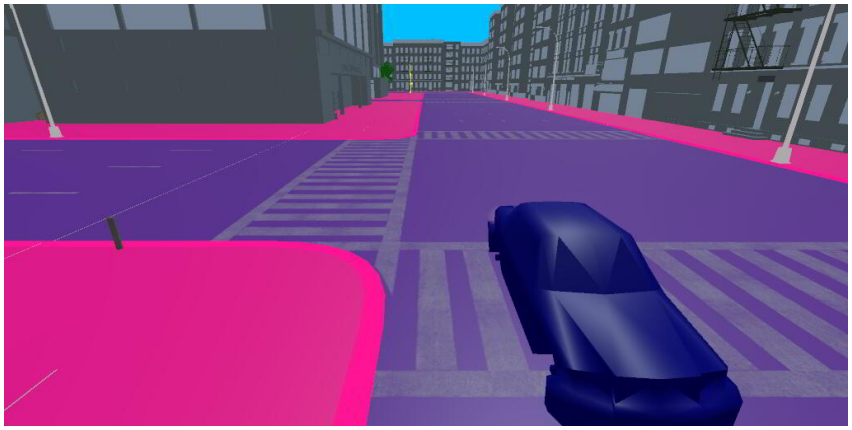


Figura 6.4: Resultado de mapa semántico *ground-truth*

La segmentación semántica en vídeo (*script* **segment_video.py**) sigue el mismo concepto que en una sola imagen, esta vez se recoge todos los cuadros en una secuencia de vídeo y lo procesa uno a uno. Después de importar los paquetes se analiza y recoge la información de los argumentos *args*, tiene la casi la misma estructura que el *script* anterior, a excepción de los siguientes 2 argumentos:

- `--video`: Se ingresa la ruta al archivo de vídeo de entrada
- `--show`: Indica si se muestra o no, el vídeo en la pantalla mientras se procesa. Se logrará un mayor de rendimiento de FPS si se establece este valor en cero.

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
  help="path to deep learning segmentation model")
ap.add_argument("-c", "--classes", required=True,
  help="path to .txt file containing class labels")
ap.add_argument("-v", "--video", required=True,
  help="path to input video file")
ap.add_argument("-o", "--output", required=True,
  help="path to output video file")
ap.add_argument("-s", "--show", type=int, default=1,
  help="whether or not to display frame to screen")
ap.add_argument("-l", "--colors", type=str,
  help="path to .txt file containing colors for labels")
ap.add_argument("-w", "--width", type=int, default=500,
  help="desired width (in pixels) of input image")
args = vars(ap.parse_args())
```

Finalmente abrir un terminal, direccionar al directorio donde esta el código y después de ejecutarlo se genera un nuevo vídeo almacenado en el directorio *output*.

6.2 Resultados Experimentales

Una vez en el directorio donde se encuentra la carpeta principal, verificamos el *script* y sus parámetros para ejecutarlo, se inserta la linea de código que se muestra a continuación y se obtiene los siguientes

resultados:

```
python segment.py --model enet-cityscapes/enet-model.net
--classes enet-cityscapes/enet-classes.txt
--colors enet-cityscapes/unity-colors.txt
--image images/example_01.png

>>[INFO] loading model...
>>[INFO] inference took 1.5242 seconds
```

Como se observa en la Figura 6.5 se presentan diferentes resultados de imágenes tomadas de las cámaras del simulador, aplicando a éstas el mapa semántico sobre los objetos (ground truth) y el algoritmo para obtener una imagen resultante.

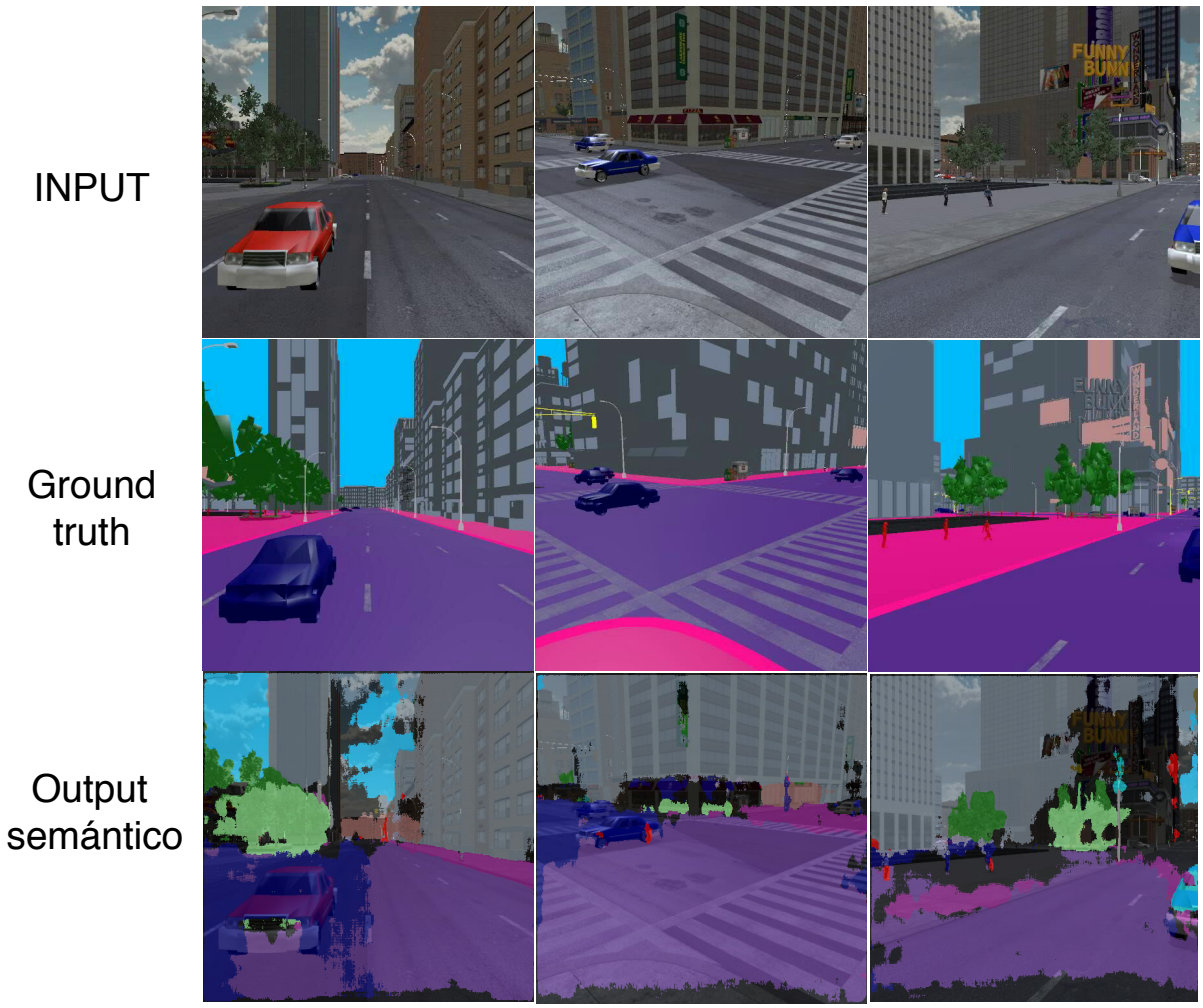


Figura 6.5: Imágenes aplicando el algoritmo (RGB | Ground truth | Segmentación Semántica)

A continuación se observa en la Figura 6.6 que en una se las cámaras que se encuentra en un objeto móvil (helicóptero), el algoritmo aplicado tiene mayor dificultad en etiquetar y segmentar los objetos de la imagen.

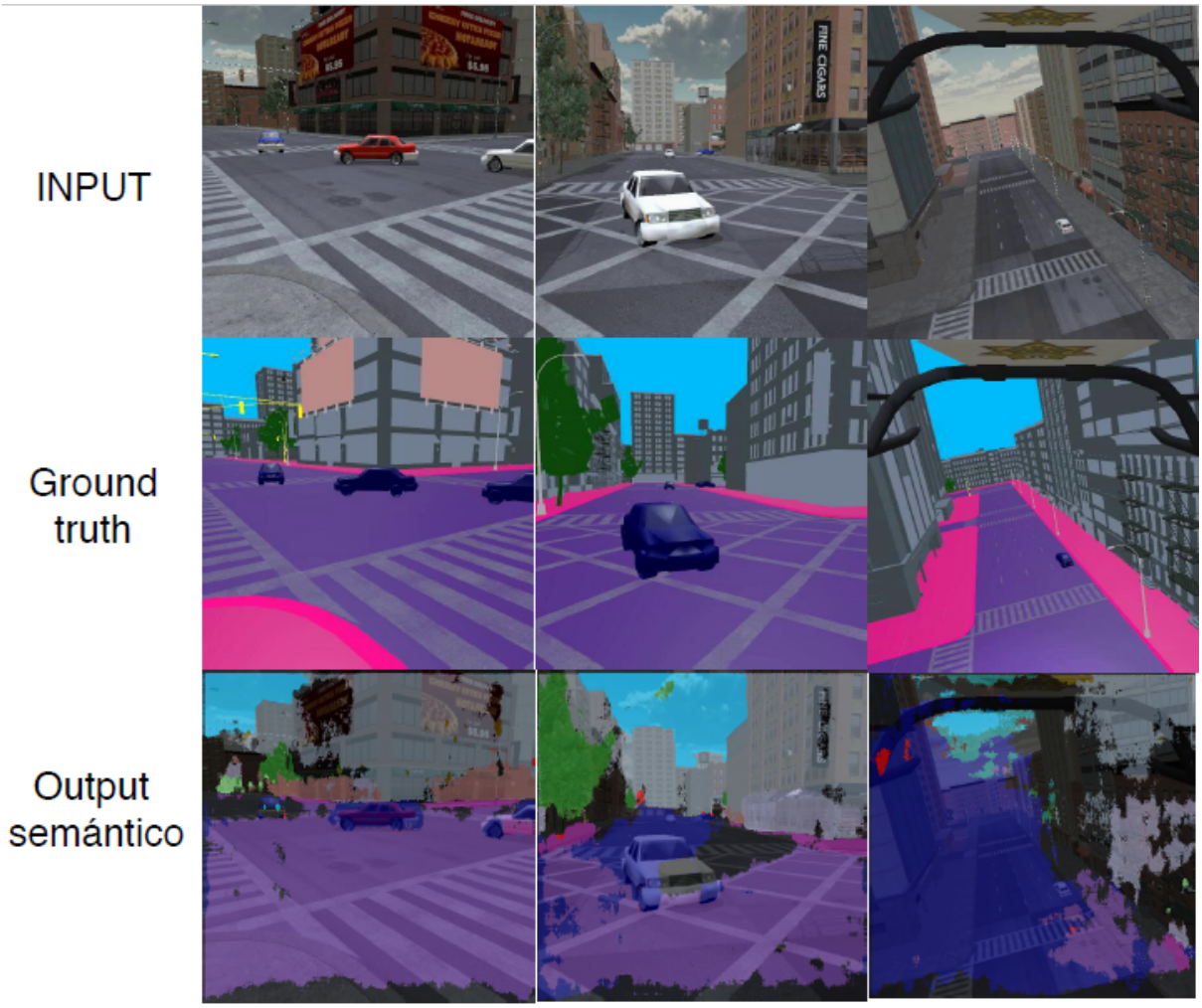


Figura 6.6: Diferentes tomas del simulador (RGB | Ground truth | Segmentación Semántica)

Otros de los resultados adicionales que se puede observar también en la Figura 6.7 es de imágenes sobre una cámara montada en este caso sobre un automóvil, igualmente la cámara instalada sobre un objeto en movimiento.

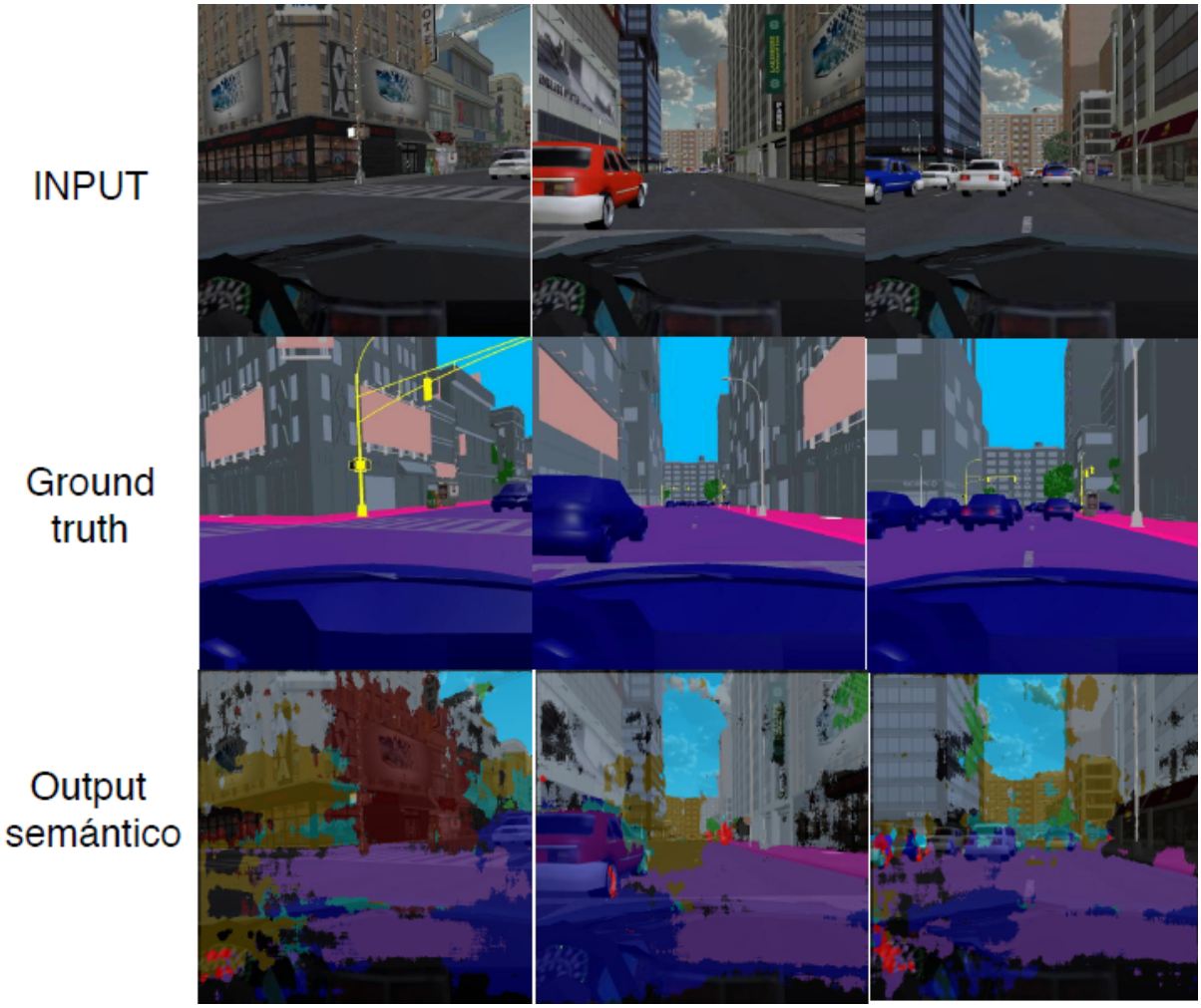


Figura 6.7: Imágenes de cámara sobre automóvil (RGB | Ground truth | Segmentación Semántica)

Para aplicar el algoritmo sobre vídeos del simulador se tiene el *script* correspondiente el cual se ejecuta con el siguiente comando:

```
python segment_video.py --model enet-cityscapes/enet-model.net
--classes enet-cityscapes/enet-classes.txt --colors
enet-cityscapes/unity-colors.txt --video videos/EJEMPLO_VIDEO_ENTRADA.avi
--output output/NOMBRE_VIDEO_SALIDA.avi

>>[INFO] loading model...
>>[INFO] 709 total frames in video
>>[INFO] single frame took 1.1819 seconds
>>[INFO] estimated total time: 837.9739
>>[INFO] cleaning up...
```

El resultado del procesamiento de un vídeo obtenido del simulador se puede observar en la Figura 6.8, donde la segmentación semántica es mas eficiente en un vídeo que en una imagen[2]. En el siguiente enlace se puede observar los experimentos de los vídeos resultantes: https://vimeo.com/407036893.

[2] www.pyimagesearch.com/2018/09/03/semantic-segmentation-with-opencv-and-deep-learning/

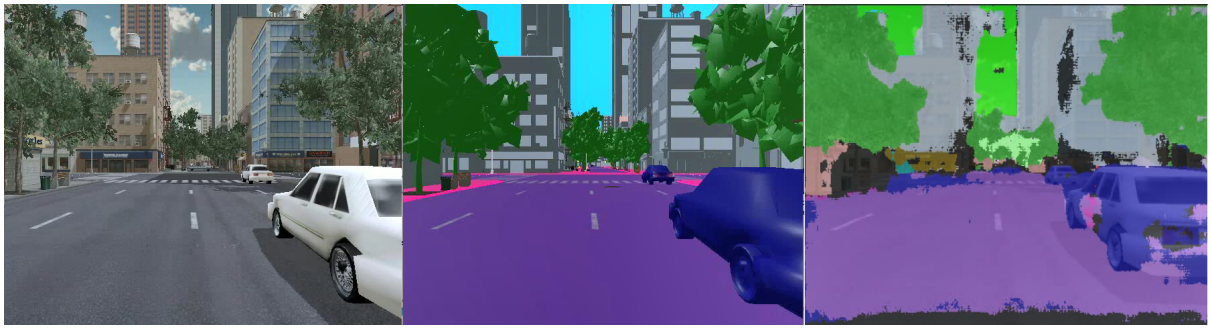


Figura 6.8: Algoritmo semántico aplicado a un vídeo del simulador

A continuación se puede observar en la Figura 6.9 mas de los resultados aplicados a distintos vídeos de cámaras ejecutándose en el simulador.

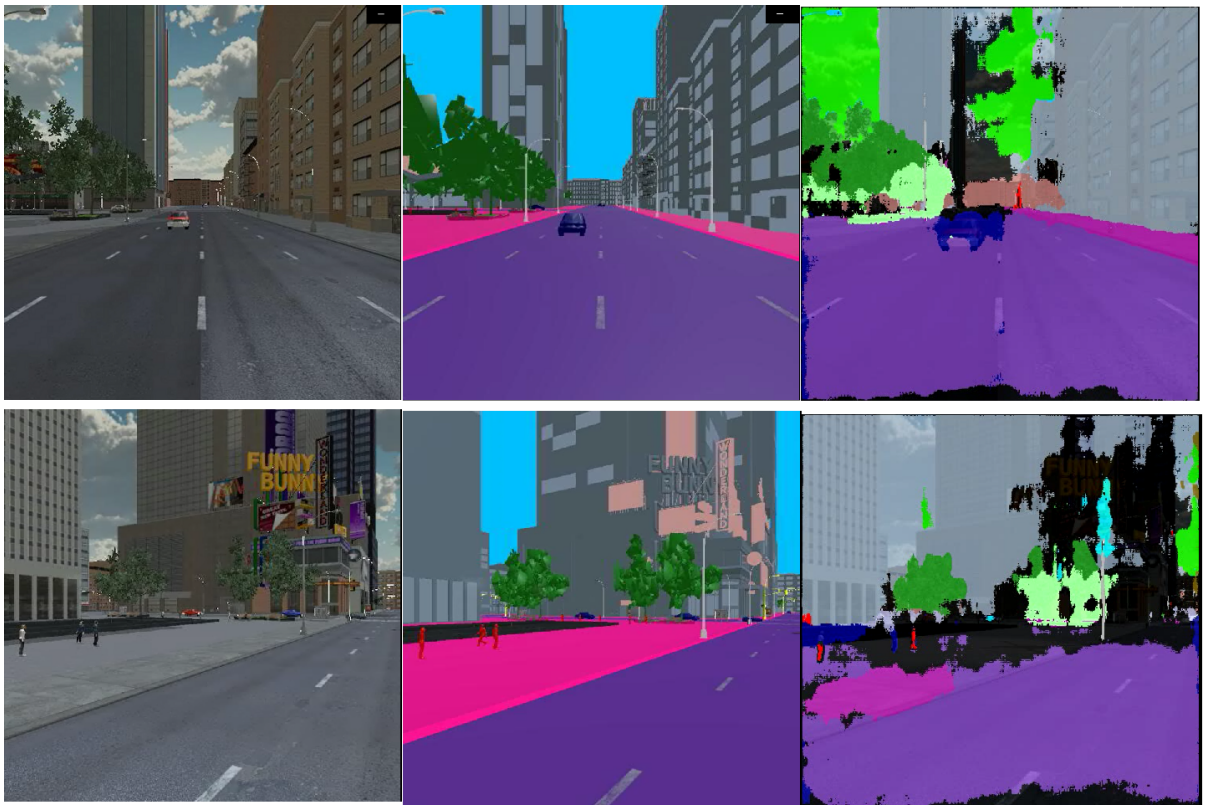


Figura 6.9: Resultados de algoritmo semántico aplicado a distintas vídeos del simulador.

Después de realizar varios experimentos con el algoritmo se puede destacar que le cuesta identificar la clase "cielo" (fondo), como se observa en las imágenes en varios casos la etiqueta que el algoritmo asigna es la de tierra ("Terrain", verde claro). Otro de los escenarios difíciles para que el algoritmo tenga un buen desempeño es cuando los objetos se encuentran muy alejados de la cámara ya que es más difícil de identificar a que clase pertenece determinado objeto.

Capítulo 7

Conclusiones y trabajo futuro

7.1 Conclusiones

- El presente documento expone a detalle el diseño e implementación de nuevas funcionalidades dentro del simulador multi-cámara teniendo como base el escenario "ciudad", proyecto en el cual se ha desarrollado gran variedad de cambios tanto en *frontend* para el usuario como en *backend* en cuanto a código y librerías entre los cuales se destaca la incorporación de cámaras de monitoreo dentro de los escenarios, permitiendo así cumplir con las mejoras del simulador.
- Una vez realizada la investigación sobre el modelo base del simulador y estudiado en el estado del arte otros desarrollos similares se puede destacar que las mejoras implementadas permiten al usuario tener una gama muy amplia de posibilidades para simular escenarios que contribuyan con las líneas de investigación de la visión artificial y procesamiento de vídeo e imágenes ya que se ofrece eventos reales replicables y sencillos de implementarlos.
- La comprensión del funcionamiento cliente y servidor del simulador ha permitido el lanzamiento de nuevos *scripts* ejemplos donde se explican a gran detalle el manejo de nuevos métodos y funciones para crear, añadir, identificar, borrar y entre muchas otras mejoras, cámaras multiuso dentro del simulador, esto representa que se puedan desarrollar tantos *scripts* como necesidades tenga el usuario para hacer simulaciones y evaluar algoritmos de inteligencia artificial para obtener resultados muy completos.
- Al poder simular eventos de interés como por ejemplo tráfico, movimiento y choques de automóviles, trayectorias de objetos dinámicos, etc, es necesario ubicar de forma adecuada y estratégica cámaras en el escenario, hecho que es posible en las mejoras de éste simulador ya que además de insertar cámaras pre-establecidas, estas cuentan con más funcionalidades tales como tipo de cámara (PTZ, móviles, estáticas) y forma de renderización al incorporar imágenes RGB y una capa semántica de clasificación de objetos, una gran ventaja para la investigación de la conducción autónoma dentro de la visión artificial. Adicionalmente la nueva API cuenta con una función que permite transmitir las imágenes de la/las cámara/s en tiempo real o bajo demanda del usuario de forma que al evaluar algoritmos con alto procesamiento no se pierda ningún *frame*.
- En el desarrollo del presente trabajo se ha definido un protocolo de evaluación el cual permite de forma estándar evaluar y capturar datos del rendimiento de los recursos de un ordenador para cualquier simulación o experimento no solo del simulador sino de cualquier aplicación futura. Se puede concluir además que una tarjeta gráfica con buenas prestaciones puede ejecutar el simulador sin mayor dificultad, ya que permite una mayor fluidez al trabajar el proyecto y las simulaciones sobre la interfaz de Unity. Con una %20 de VRAM (algo menos de 2GB), se obtiene buenas resultados para trabajar con vídeos de resolución alta, estos datos compaginan con pruebas de rendimiento para videojuegos en 3D.

- La usabilidad de la GPU es baja comparando con el rendimiento de la CPU en la cual existe un cuello de botella al utilizar el simulador y la API del cliente en el mismo ordenador, sin embargo es esperable ya que una tarjeta gráfica dedicada esta especialmente diseñada para correr videojuegos muy exigentes computacionalmente y proyectar imágenes de resoluciones cercanas a 2K y 4K, al igual que la VRAM permite almacenar texturas y mucho detalle para mostrarlas en pantalla en tiempo real y una latencia muy baja. Con estos aspectos se puede concluir que con los detalles, texturas y calidad de resolución del diseño realizado sobre el simulador es muy poco probable que se llegue a saturar la GPU, por lo cual el rendimiento sobre la CPU va a tener un mayor peso en los resultados presentados.

7.2 Trabajo futuro

Los análisis de resultados permiten definir y proponer mayores alternativas en las que se pueden trabajar hecho por el cual se propone ampliar la investigación en los siguientes puntos:

- Profundizar en el diseño de nuevos entornos e incorporar el proyecto mas objetos 3D que simulen el mundo real, es decir construir más tipos de ciudades y objetos dinámicos que permitan simular escenarios mucho mas reales y complejos, dentro de los cuales se pueda automatizar la identificación y clasificación de objetos de forma que se amplíe las capas dentro de la segmentación semántica.
- Obtener una base de datos amplia de vídeos del simulador con los cuales se pueda entrenar una red neuronal en ciudades y exteriores para probar y comparar otros algoritmos enfocados con la segmentación semántica y la conducción autónoma de forma que se puedan recrear eventos controlados dentro del simulador. Esto permite así buscar nuevas alternativas y mejoras en algoritmos dentro del campo de la inteligencia artificial.

Bibliografía

[1] J. Peddie, K. Akeley, P. Debevec, E. Fonseka, M. Mangan, and M. Raphael, "A vision for computer vision: emerging technologies," in *ACM SIGGRAPH 2016 Panels*, pp. 1–2, 07 2016. 1

[2] W. Starzyk and F. Qureshi, "Software laboratory for camera networks research," *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, vol. 3, pp. 284–293, 06 2013. 1, 2

[3] W. Burger, Matthew J. Barth, and W. Sturzlinger, *Immersive Simulation for Computer Vision*. Joint 19th AGM and 1st SDRV workshop Visual Modules, 1995. 1

[4] F. L. García, "Desarrollo de escenarios para simulador multi-cámara basado en unity," Master's thesis, Escuela Politecnica Superior (Universidad Autonoma de Madrid), Junio 2018. 1, 2, 9, 10

[5] H. Hattori, V. Naresh Boddeti, K. M. Kitani, and T. Kanade, "Learning scene-specific pedestrian detectors without real data," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. 1

[6] César R. de Souza, A. Gaidon, Y. Cabon, and Antonio M. López, "Procedural generation of videos to train deep action recognition networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–20, Julio 2017. 1

[7] Mario González Jiménez, "Sistema multi-cámara distribuido basado en Unity," in *Trabajo final de grado, Escuela Politécnica Superior (Universidad Autonóma de Madrid)*, Enero 2017. 2, 4, 7, 8, 9, 10, 12, 13, 18, 19

[8] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Junio 2016. 2, 3

[9] F. Z. Qureshi and D. Terzopoulos, "Surveillance in virtual reality: System design and multi-camera control," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, June 2007. 2

[10] X. Wang, "Intelligent multi-camera video surveillance: A review," in *Pattern Recognition Letters*, vol. 34, pp. 3–19, 2013. 2

[11] Juan C. San Miguel, José M. Martínez, and A. García, "An ontology for event detection and its application in surveillance video," in *2009 Advanced Video and Signal Based Surveillance*, Septiembre 2009. 2, 8

[12] D. Biedermann, M. Ochs, and R. Mester, "Congrats: Realistic simulation of traffic sequences for autonomous driving," in *IVCNZ 2015 Image and Vision Computing New ZealandAt*, 11 2015. 2, 8

[13] V. Veeravasarapu, C. Rothkopf, and V. Ramesh, "Model-driven simulations for computer vision," in *2017 IEEE Winter Conference on Applications of Computer Vision*, pp. 1 – 9, Marzo 2017. 2

[14] W. Qiu and A. Yuille, "Unrealcv: Connecting computer vision to unreal engine," *arXiv preprint arXiv:1609.01326*, pp. 1–8, 09 2016. 2

[15] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. M. Lopez, "The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3234–3243, June 2016. 3

[16] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 740–755, Springer International Publishing, 2014. 3

[17] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "Enet: A deep neural network architecture for real-time semantic segmentation," *CoRR*, vol. abs/1606.02147, 2016. 4, 55, 56

[18] U. Technologies, *Unity Manual Documentation.* Copyright © 2020 Unity Technologies. Version 2019.3. 8

[19] I. Ouazzani, "Manual de creación de videojuego con unity 3d," Master's thesis, Universidad Carlos III de Madrid, Agosto 2012. 8

[20] M. Cordts, M. Omran, S. Ramos, T. Scharwächter, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset," in *CVPR Workshop on The Future of Datasets in Vision*, 2015. 23, 24

[21] H. Zhao, X. Qi, X. Shen, J. Shi, and J. Jia, "Icnet for real-time semantic segmentation on high-resolution images," in *Computer Vision – ECCV 2018* (V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, eds.), (Cham), pp. 418–434, Springer International Publishing, 2018. 23

Apéndice

Apéndice A

Operar API de Python

A.1 Preparación de Software

Partiendo de la premisa de haber descargado e instalado con éxito Ubuntu en una máquina virtual o tener como sistema operativo principal linux/ubuntu en su versión 18.04.4 LTS, se puede encontrar Python preinstalado en sus dos versiones 2.x y 3.x, con este último es el cual se trabajará la API he indicarán todas las configuraciones dentro de este documento. Para conocer las versiones que están instaladas en el sistema, puede hacerlo a través de los siguientes comandos, para Python 2 y 3 respectivamente, como puede ver en la Figura A.1

```
~$ python -V
~$ python --version
~$ python3 -V
~$ python3 --version
```

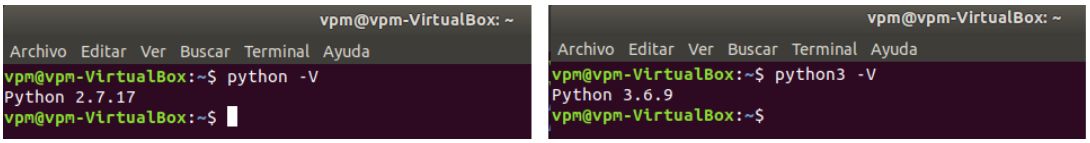


Figura A.1: Versiones de Python

A.2 Entorno de Trabajo

Para la API del simulador se trabajará sobre Python 3, por lo cual es necesario crear un nuevo entorno de trabajo con la versión de Python 3.6.9. Se instalará y actualizara, como primer paso, la librería *pip* de Python para posteriormente instalar todos los paquetes necesarios dentro del entorno de trabajo.

```
~$ sudo apt-get install python3-pip
```

Se suele presentar algún tipo de problemas en ubuntu para instalar la librería mencionada por lo cual es necesario realizar una configuración previa al sistema, mediante el comando a continuación se soluciona, sin embargo si no existe ningún impedimento al instalar *pip* esta paso se lo puede omitir.

```
~$ sudo dpkg --configure -a
```

Posteriormente se hace la instalación de *virtualenv* la cual permitirá crear tantos entornos como versiones de Python distintas se desee usar, estos diferentes entornos son independientes unos de otros y todas las versiones de paquetes instaladas dentro de alguna no afecta a las demás.

```
~$ sudo apt-get install python-virtualenv virtualenv
```

- **virtualenv con Python 3**
  Para crear un entorno virtual con Python 3, simplemente ejecutamos el comando virtualenv de la siguiente manera:

```
~$ virtualenv env --python=python3
```

- **virtualenv con Python 2**
  Para crear un entorno virtual con Python 2, simplemente ejecutamos el comando virtualenv de la siguiente manera:

```
# Debian, Ubuntu, CentOS o Fedora
~$ virtualenv env
```

- **Activar un entorno virtual de Python con virtualenv**
  Para activar un entorno virtual de Python, se ejecuta el *script activate* de virtualenv instalado en el directorio bin/: (dentro de la carpeta que se a creado de nombre "env")

```
 ~$ cd env
 ~$ source bin/activate
(env)~$ # within the environment
```

- **Desactivar un entorno virtual de Python con virtualenv**
  Para desactivar un entorno virtual, porque se necesita trabajar en otro diferente, se ejecuta el comando *deactivate* de virtualenv. No es necesario ir al directorio del entorno virtual para realizar esta operación:

```
(env)~$ deactivate
```

Dentro de este entorno con la versión de Python deseada, en este caso 3.6.8, es necesario instalar algunas librerías importantes para el funcionamiento de la API, en el archivo de *requirements.txt* se enumeran las principales que se han precedido a instalar, recordar que se debe estar en el entorno deseado para su ejecución al igual que ir al directorio donde se encuentra el fichero, observar Figura A.2

```
(env)~$ pip install -r requirements.txt
```

```
# Instalar ejecutando "pip install -r requirements.txt"
scikit-image==0.15.0
scipy==1.3.0
memory_profiler==0.55.0
opencv-python==4.1.0.25
tensorflow==1.14.0
jupyter==1.0.0
pandas==0.24.2
setuptools==41.0.0
imutils

# paquetes para segmentacion semantica
cycler==0.10.0
kiwisolver==1.0.1
matplotlib==3.0.2
numpy==1.16.0
Pillow==6.2.0
pyparsing==2.3.1
python-dateutil==2.7.5
pytz==2018.9
six==1.12.0
torch==1.0.0
torchvision==0.2.1
```

```
vpm@vpm-VirtualBox: ~/env
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
vpm@vpm-VirtualBox:~$ cd env/
vpm@vpm-VirtualBox:~/env$ ls
bin  CodesPython  etc  include  lib  requirements.txt  share
vpm@vpm-VirtualBox:~/env$ source bin/activate
(env) vpm@vpm-VirtualBox:~/env$ ls
bin  CodesPython  etc  include  lib  requirements.txt  share
(env) vpm@vpm-VirtualBox:~/env$ python -V
Python 3.6.8
(env) vpm@vpm-VirtualBox:~/env$ pip list
Package              Version
-------------------- --------
absl-py              0.8.0
astor                0.8.0
attrs                19.1.0
backcall             0.1.0
bleach               3.1.0
```

Figura A.2: Librerías requeridas e instaladas

A.3 Funciones y librerías de la API

La API implementada con Python contiene las siguientes clases y funciones:

Tabla A.1: Funciones de MSScam.py.

Función	**Descripción**
__init__(self,*args): Default(self): Manual(self,*args): init(self):	Clase Constructor, parámetros por defecto, parámetros específicos, parámetros cargados desde un archivo texto
ManualName(self,*args):	Método para asignar distinto nombre al objeto de una conexión cliente, y acceder a cámaras de distinto nombre
initFromDescriptorExtended(self,values):	Método para inicializar los parámetros de la cámara desde una cadena recibida del simulador. Ver más en MSSclient.getAllCamerasFromSimulator.
initiate_from_camera_descriptor(self,values):	Método para inicializar los parámetros de la cámara desde una cadena recibida del simulador. Ver más en MSSclient.getAllCamerasFromSimulator.
saveTofile(self,filename):	Método para guardar los parámetros de la cámara en un archivo de configuración. Actualmente es compatible con una descripción por archivo.
setRegister(self,sock,ipAddress,port):	Hacer el registro de un objeto "objectCamera" en el simulador sin crear necesariamente el objeto en el simulador

setNameRACamera(self,newNameRACamera):	Método para asignar un nombre al objeto de una conexión cliente, y acceder a cámaras de Acceso Remoto (RA) del simulador según el nombre designado al registrar las cámaras en el servidor
configRACamera(self,sock,ipAddress,port):	Configurar a las cámaras pre-establecidas en el simulador con parámetros definidos que envía el cliente
readTosimulator(self,sock,ipAddress,port):	Identificar el numero de cámaras de Acceso Remoto que están pre-establecidas en el simulador
addToSimulatorMapaSemantico(self,sock,ipAddress,port)	Añadir cámaras al simulador incluyendo la capa de mapa semántico para su visualización
addTosimulator(self,sock,ipAddress,port):	Añadir cámaras al simulador
removeCameraSceneSimulator(self):	Eliminar una cámara de Acceso Remoto del simulador (cámaras que se establecieron previamente dentro del simulador en un escenario en concreto)
removeFromSimulator(self):	Remover la cámara del simulador (creada por el mismo cliente)
operator(self):	Obtener el *frame* actual del simulador
receiveFrame(self,verbose):	Esta función lee grandes cantidades de datos empleando un pequeño buffer de 10 KB
convertBufferToMat(self):	Decodifica el buffer interno en una variable cv :: Mat. Utiliza la API de OpenCV, por lo que se admiten formatos de imagen estándar.
preview(self,timeout):	Muestre cuadros continuamente hasta que se presione la tecla ESC o se active el tiempo de espera.
setStatus(self,connected):	Establecer el estado de la cámara
realFPSenable(self):	Habilitar el recuento de framerate en el servidor
realFPSdisable(self):	Deshabilitar el recuento de framerate en el servidor
updateRealFPS(self):	Recuperar recuento de framerate del servidor
move(self,option):	Método para mover la cámara en el simulador.
setFPS(self,newfps):	Método para establecer la velocidad de fotogramas de la cámara y actualizarla en el servidor MSS
setTXRXformat(self,newfmt):	Método para configurar el formato TX / RX para la cámara y actualizarlo en el servidor MSS
setJPEGquality(self,newq):	Método para configurar la calidad JPEG de la cámara y actualizarla en el servidor MSS
zoomIn(self):	Método para aumentar el zoom de la cámara y actualizarlo en el servidor MSS

zoomOut(self):	Método para disminuir el zoom de la cámara y actualizarlo en el servidor MSS
setWidth(self,n_width):	Método para establecer el ancho de los marcos capturados y actualizarlo en el servidor MSS
setHeight(self,n_height):	Método para establecer la altura de los fotogramas capturados y actualizarlo en el servidor MSS
send_command(self,command_in,reply_in,verbose):	Método para enviar un comando al servidor MSS y recibir una respuesta (la mayoría de los ACK)
send_command_noACK(self,command_in,verbose):	Método para enviar un comando al servidor MSS y no recibir una respuesta (la mayoría de los ACK)
print_details(self):	Método para mostrar detalles de la cámara en la ventana de comandos
GUIcontrol(self):	Método para iniciar el control interactivo del movimiento de la cámara a través de una GUI
GUIinsert(self,sock):	Método para insertar de forma interactiva del movimiento de la cámara a través de una GUI
GUIControlRACam(self):	Método controlar el movimiento de las cámaras de acceso Remoto (RA) a través de una GUI

Tabla A.2: Funciones de MSScam_control.py.

Función	**Descripción**
start(self,cam):	Método para iniciar el movimiento interactivo de la cámara.
moveCam_UpDown(self,value):	Método de devolución de llamada para mover la cámara UP & DOWN (Y-axis)
moveCam_LeftRight(self,value):	Método de devolución de llamada para mover la cámara LEFT & RIGHT (X-axis)
moveCam_AheadBack(self,value):	Método de devolución de llamada para mover la cámara FORWARD & BACKWARD (Z-axis)
moveCam_rotateLeftRight(self,value):	Método de devolución de llamada para rotar la cámara LEFT & RIGHT (X-axis)
moveCam_rotateUpDown(self,value):	Método de devolución de llamada para rotar la cámara UP & DOWN (Y-axis)

Tabla A.3: Archivos de Python que contiene MSScam_insert.py

class MiniMapInfo:	**Descripción**
__init__(self):	self.createMiniMapInfo()
createMiniMapInfo(self):	inicializa MiniMap
OnClickMap(event, x, y, flags,data):	Método de devolución de llamada para detectar eventos CLIC en el mapa visualizado de la escena (vista cenital). Obtiene la ubicación X-Z de la cámara en la simulación MSS.
OnClickHeightMap(event, x, y, flags, data):	Método de devolución de llamada para detectar eventos CLIC en el mapa visualizado de la escena (vista lateral). Obtiene la altura (eje Y) de la cámara en la simulación MSS.
class MSScam_insert:	**Descripción**
__init__(self):	Constructor de clase predeterminado con parámetros predeterminados
start(self, socket, cam_in):	Método para iniciar la inserción interactiva de la cámara.
receiveMap(self,name, verbose):	Método para leer una imagen proporcionada por el servidor MSS que describe el escenario (vista cenital)
coordinateConversion(self,mini, maxi, pointRelative):	Método para convertir entre un punto seleccionado en coordenadas locales (ancho, alto) y un punto en coordenadas de Unidad mundial (XYZ)
getMiniMapInfoServer(self, sock):	Esta función recibe de MSSserver información que se utilizará para cámaras de inserción visual.
showMiniMap(self):	Método para mostrar el minimapa (vista de escena cenital) y superpone un icono de cámara para visualizar la ubicación de la cámara.
showCameraScheme(self):	Esta función muestra el mapa lateral (vista de escena lateral) y superpone un icono de cámara para visualizar la ubicación de la cámara (altura).

Tabla A.4: Funciones de MSSclient.py.

Función	**Descripción**
__init__(self):	Inicialización del cliente predeterminado
initialize_network(self):	Inicialización de la configuración de red para UNIX, devuelve Integer> 0 si está bien (-1 si falla)
connectTosimulator(self,ipAddress,port,sock):	Esta función debe ser la primera utilizada. Crea el socket.
disconnectFromSimulator(self,sock):	Esta función debe usarse para desconectarse del simulador. De lo contrario, un cliente 'dead' permanece en el simulador.
resetSimulator(self,sock):	Esta función reinicia el simulador
getAllCamerasFromSimulator(self,sock):	Obtiene los datos de las cámaras conectadas al simulador.

Apéndice B

Versiones de Python

B.1 De Python 2.X a 3.X

Inicialmente, la API de Python tenía una versión 2.7.X, por lo que se ha migrado a una versión más actual de Python, específicamente 3.6.9, esto implica que hay líneas de código incompatibles y bibliotecas obsoletas que presentarán un error en ese momento. de su ejecución, los principales cambios realizados en la nueva versión de la API se indican a continuación.

```
print("This demo test: \n") # Python 3.X incorporates parentheses to print
print "This demo test:\n"   # Python 2.x
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

self.id = int(reply.rstrip('\x00')) # Python 3.X replace replace instruction
                                    # to remove the "'\ x00'" if used for string,
                                    # if it is between bytes use "b '\ x00'"
#self.id = int(reply)               #Python2.x this value is obtained the int
                                    # of the reply variable
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

chars_array=smallbuf.split(b'#')    # Python 3.x divide by bytes
#chars_array=smallbuf.split("#")    # Python 2.x split one byte
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

buf=buf+b'='*(length4-len(buf))     # Python 3.x concatenacion of byte type buf
                                    # with b '=' byte type
#buf=buf+"="*(length4-len(buf))     # Python 2.x TypeError: can't concat str to
                                    # bytes, buf is type byte and "=" type string
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
s=base64.b64decode(buf)             # Python 3.x decode on base 64
#s=buf.decode('base64')             # Python 2.x syntax in that version

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
if self.sock.send(command.encode('utf-8'), 0) < 0:  # Python 3.x you need to
                                                    # send the command of type
                                                    # byte and not string
#if self.sock.send(command, 0) < 0:             # Pyton2.x send without decoding
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
command=str(command,'utf-8')        # Python 3.x it's becoming a string
#command=command.replace("\0","")   # Python 2.x b'1 \ x00 'is being replaced
                                    # by empty space to count
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
35 if self.sock.send(command.encode('utf-8'), 0) < 0:  # Python 3.x you need to
36                                                     # send the command of type
37                                                     # byte and not string
38 #if self.sock.send(command, 0) < 0:          #Python 2.x TypeError: a bytes-like
39                                              # object is required, not 'str'
40 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
41
42 self.gui=MSScam_insert()                    # Python 3.x
43 #self.gui=MSScam_insert.MSScam_insert() # Python 2.x Error:type object 'MSScam_
44                                          # insert' has no attribute 'MSScam_insert
       '
45 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
46 self.rotZ = float(values[11].rstrip('\x00'))     # Python3.x, .rstrip -> delete
47                                                  # the final characters
48
49 # self.rotZ = float(values[11])                 # Python 2.x values is type
50 # <class 'list'> y la lista es [.., '10', '0\x00'] where the value [11] is byte
51 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
52
53 message=message.rstrip(b'\0')      #Python3.x remove the last characters of'\x00'
54 #message=message.replace("\0","") #instruction of Python 2.x
55 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
56
57 chars_array = message.split(b'/')   #Python3.x indicates to split between bytes
58 #chars_array = message.split("/")   #TypeError: a bytes-like object is required,
59                                     #not 'str' Python 2.x can do between byte and
60                                     # string
61 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
62
63 s = base64.b64decode(buf)    # Python 3.x decode in base64
64 #s = buf.decode('base64')    # syntax for Python 2.x
65 ____________________________________________________________________________
```

Listing B.1: Versiones de Python

Apéndice C

Códigos para evaluación

C.1 Código Matlab para generar resultados

El código que se presenta a continuación permite sintetizar los datos y presentarlos en figuras para realizar un análisis cualitativo, esto a partir de un archivo plano de texto (.txt) donde se encuentra los datos recolectados del rendimiento del ordenador.

```
% Ejemplo para generar BOXPLOT desde un archivo txt
%
% Juan Carlos San Miguel
% Universidad Autonoma de Madrid
% 2016/10/19

% Editor
% Vinicio Pazmino
% 2020/05/27

close all
clear
clc;
outconf = 10;
fprintf('Rango de confianza para la eliminacion de valores atipicos: [%.2f %.2f ]\n\n',outconf,100-outconf)

% Cargar Datos
fileID = fopen('RendimientoAgrupados.txt');
data = textscan(fileID,'%f %f %f %f %s');
parametro=1; % 1=CPU 2=GPU 3=RAM 4=VRAM elegir la columna de datos a graficar

% Encontrar etiquetas unicas y enlistar
res = unique(data{5},'stable')
res_ordered = res

% Recuperar y trazar datos numericos para cada resolucion
figure;
hold on;
for i = 1:numel(res)
    timeRes{i} = filter_data(data, res_ordered{i}, parametro);
    fprintf('Resolution %s: %d values\n',res_ordered{i},numel(timeRes{i}));
    labels = bplot(timeRes{i},i,'whisker',outconf);
end

```

```
35 title('Evaluacion de Rendimiento del Simulador')
36 legend(labels(1:end-1),'Location','northwest'); %legend for plotted data
37 xlabel('Resoluciones frente a Camaras Usuario (RGB y Semantico)'); %texto para
       X label
38 set(gca,'XTick',1:numel(res),'XTickLabel', res_ordered); %texto para cada
       elemento de X label
39 ylabel('Uso de la CPU(%)'); %label eje Y
40
41 % set(gca,'yscale','log'); % cambiar a escala logaritmica (a veces mejora la
       visualizacion)
42 axis([0.5 numel(res)+0.5 0 100]); % ajustar los limites del eje
43
44 % Guardar figura
45 fileformat = {'epsc','png'};
46
47 for i=1:numel(fileformat)
48     fprintf('\nSaving boxplot to format %s...',fileformat{i});
49     saveas(gca, 'figure',fileformat{i});
50 end
51 fprintf('Done.');
```

Listing C.1: Código de Matlab para graficar resultados de rendimiento (Archivo Main)

El archivo *Main* hace uso de dos ficheros funciones de Matlab los cuales son "filter_data.m" y "bplot.m"

```
1  function result = filter_data(data, label, parametroRendimiento)
2  % Editor
3  % Vinicio Pazmino
4  % 2020/05/27
5  % Seleccionar los elementos de determinado parametro
6  % parametroRendimiento = 1 (Columna 1 de txt = usoCPU)
7  % parametroRendimiento = 2 (Columna 2 de txt = Ram)
8  % label = los datos que tengan la misma etiquete (resoluciones de 320x240)
9  %         segun la columna 5 del txt
10     result=[];
11     for i=1:numel(data{5})
12
13         if strcmp(data{5}{i},label)
14             result = [result data{parametroRendimiento}(i)];
15         end
16     end
17 end
```

Listing C.2: Funcion adicional de Matlab (función "filter_data.m")

Apéndice D

Preparación de algoritmo: Segmentación Semántica

D.1 Algoritmo Semántico: Imagen

```
# load the class label names
CLASSES = open(args["classes"]).read().strip().split("\n")

# if a colors file was supplied, load it from disk
if args["colors"]:
  COLORS = open(args["colors"]).read().strip().split("\n")
  COLORS = [np.array(c.split(",")).astype("int") for c in COLORS]
  COLORS = np.array(COLORS, dtype="uint8")

  np.random.seed(42)
  COLORS = np.random.randint(0, 255, size=(len(CLASSES) - 1, 3),
    dtype="uint8")
  COLORS = np.vstack([[0, 0, 0], COLORS]).astype("uint8")
```

Se carga `CLASSES` en la memoria desde el archivo de texto suministrado donde la ruta esta en la linea de comando `args` diccionario (**linea 2**). Si un conjunto con especificaciones previas `COLORS` para cada etiqueta de clase se proporciona en el archivo de texto (uno por linea), se los carga en la memoria. De lo contrario, se genera colores aleatoriamente por cada etiqueta.

Posteriormente se crea una leyenda (con las 20 clases) de búsqueda de color usando las funciones de dibujo de OpenCV, se genera visualmente un panel para asociar fácilmente una etiqueta de clase con un determinado color. La leyenda (**linea 2**) se construye dinámicamente mediante el bucle (**linea 5-11**).

```
# initialize the legend visualization
legend = np.zeros(((len(CLASSES) * 25) + 25, 300, 3), dtype="uint8
    ")

# loop over the class names + colors
for (i, (className, color)) in enumerate(zip(CLASSES, COLORS)):
  # draw the class name + color on the legend
```

```
7     color = [int(c) for c in color]
8     cv2.putText(legend, className, (5, (i * 25) + 17),
9       cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
10    cv2.rectangle(legend, (100, (i * 25)), (300, (i * 25) + 25),
11      tuple(color), -1)
```

La segmentación semántica se realiza en el siguiente bloque. Cargar el modelo (**linea 3**), Construir un `blob` (**linea 8-11**). El modelo ENet de este ejemplo se ha entrenado con una resolución de 1024x512. Se selecciona el `blob` como entrada a la red (**linea 14**) y se realiza un paso directo a través de la red neuronal (**linea 16**). Marca de tiempo transcurrido se observa en la (**linea 20**).

```
1  # load our serialized model from disk
2  print("[INFO] loading model...")
3  net = cv2.dnn.readNet(args["model"])
4
5  # load the input image, resize it, and construct a blob from it,
6  # but keeping mind mind that the original input image dimensions
7  # ENet was trained on was 1024x512
8  image = cv2.imread(args["image"])
9  image = imutils.resize(image, width=args["width"])
10 blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (1024, 512), 0,
11   swapRB=True, crop=False)
12
13 # perform a forward pass using the segmentation model
14 net.setInput(blob)
15 start = time.time()
16 output = net.forward()
17 end = time.time()
18
19 # show the amount of time inference took
20 print("[INFO] inference took {:.4f} seconds".format(end - start))
```

A continuación se genera un mapa de color para superponer en la imagen original. Cada píxel tiene un índice de etiqueta correspondiente a su clase, se extrae información de dimensión de volumen de la salida, seguido se calcula el mapa de clase y la máscara de color.
Se determina las dimensiones de la salida **(linea 1)**, se realiza una búsqueda del índice de etiqueta de la clase con la mayor probabilidad para todas las coordenadas (x,y) **(linea 7)**, esto se conoce ahora como `classMap` y contiene un índice de clase para cada píxel.

Dados los índices de ID de las clases se usa la indexación de matriz NumPy para buscar el color de visualización correspondiente para cada píxel **(linea 11)**, la mascara se superpone con una determinada transparencia a la imagen original.

```
1  (numClasses, height, width) = output.shape[1:4]
2
3  # our output class ID map will be num_classes x height x width in
4  # size, so we take the argmax to find the class label with the
5  # largest probability for each and every (x, y)-coordinate in the
```

```
# image
classMap = np.argmax(output[0], axis=0)

# given the class ID map, we can map each of the class IDs to its
# corresponding color
mask = COLORS[classMap]
```

Se dimensiona la `Mask` y el `classMap` de modo que las dimensiones sean exactamente iguales **(linea 2-5)**, es fundamental que se aplique la interpolación de vecino mas cercano en lugar de interpolación cubica, bicubica, etc., ya que se desea mantener los valores original de clases IDs/mask. Ya con el tamaño correcto se crea una "superposición de color transparente" superponiendo la mascara en la imagen original **(linea 9)**. Finalmente las imágenes resultado se visualizan en pantalla **(linea 12-14)**.

```
# case you wanted to extract specific pixels/classes)
mask = cv2.resize(mask, (image.shape[1], image.shape[0]),
  interpolation=cv2.INTER_NEAREST)
classMap = cv2.resize(classMap, (image.shape[1], image.shape[0]),
  interpolation=cv2.INTER_NEAREST)

# perform a weighted combination of the input image with the mask
   to
# form an output visualization
output = ((0.4 * image) + (0.6 * mask)).astype("uint8")

# show the input and output images
cv2.imshow("Legend", legend)
cv2.imshow("Input", image)
cv2.imshow("Output", output)
cv2.waitKey(0)
```

D.2 Algoritmo Semántico: Vídeo

Para un funcionamiento adecuado se debe importar las librerías necesarias para la ejecución:

```
import numpy as np
import argparse
import imutils
import time
import cv2
```

Se abre un puntero de flujo para ingresar el archivo e inicializarlo la grabadora de vídeo **(linea 1-2)**. Se intenta determinar el total de cuadros del vídeo y con ese valor se aproxima a calcular el tiempo de ejecución del archivo Python para procesar el vídeo **(linea 5-9)**.

```
vs = cv2.VideoCapture(args["video"])
writer = None
```

```
3
4  # try to determine the total number of frames in the video file
5  try:
6    prop =  cv2.cv.CV_CAP_PROP_FRAME_COUNT if imutils.is_cv2() \
7      else cv2.CAP_PROP_FRAME_COUNT
8    total = int(vs.get(prop))
9    print("[INFO] {} total frames in video".format(total))
```

El ultimo bloque del código se verifica se se debe mostrar los *frame* de salida, mientras se muestra los cuadros en una pantalla, si se presiona **"q"** se saldrá del bucle de procesamientos de imágenes **(linea 4-8)**, y al final se limpia los punteros.

```
1    # check to see if we should display the output frame to our
      screen
2    if args["show"] > 0:
3      cv2.imshow("Frame", output)
4      key = cv2.waitKey(1) & 0xFF
5
6      # if the `q` key was pressed, break from the loop
7      if key == ord("q"):
8        break
9
10 # release the file pointers
11 print("[INFO] cleaning up...")
12 writer.release()
13 vs.release()
```